Základní modely životního cyklu software, softwarový proces, metodika.

Wednesday, May 29, 2013 4:49 PM

Proces – systematická série akcí vedoucí k určitému výsledku

Životní cyklus (meta proces?) – proces od zahájení vývoje až po vyřazení z provozu

Metodika

definovaný proces pro konkrétní účel, tj. fáze, aktivity, role, artefakty, milníky atd. jsou dobře popsány (z přednášky)

- souhrn doporučených praktik a postupů, pokrývajících celý životní cyklus vytvářené aplikace

Z <http://cs.wikipedia.org/wiki/Metodika>
  - Booch method
  - SSADM
  - Rational Unified Process
  - SCRUM
  - …
- UML není metodika!

Softwarový proces

systematická série fází a **aktivit**, souvisejících **rolí** a **artefaktů** vedoucí k vyšší pravděpodobnosti úspěšného vytvoření potřebného software.

- Výsledek = kvalitní software
- Členění: fáze, **aktivity**
  - *Technické aktivity: komunikace, plánování, modelování, konstrukce, nasazení*
  - *Podpůrné aktivity: řízení, kontrola kvality, správa konfigurace, dokumentace*
- Mezivýsledky: **artefakty**
  - *Technické: specifikace, dokumentace, testy, modely, …*
  - *Komunikační: specifikace, plán*
  - *Obchodní: plán, rozpočet, produkt*
- Činitelé: **role**
  - *Technické role: analytik, architekt, vývojář, tester, databázista, …*
  - *Manažerské role: team leader, šéf vývojářů, šéf projektů, CEO, CIO, …*
  - *Podpůrné role: poradce, lektor, uživ. podpora, dokumentace, …*

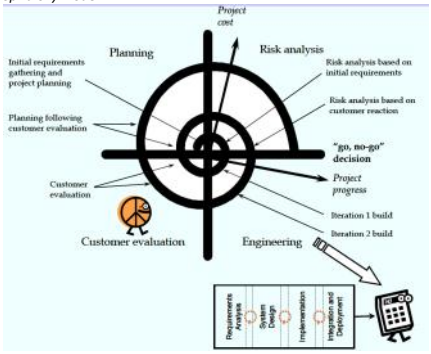
Varianty SW procesu (modely životního cyklu software):

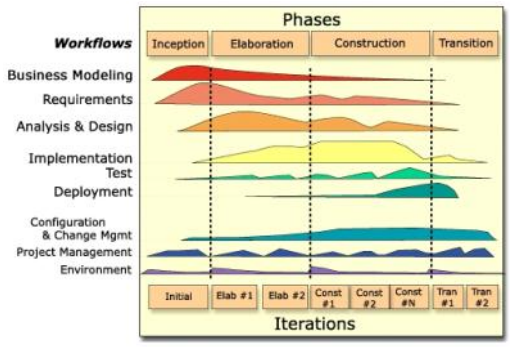
Varianty a příklady procesů:

Společná snaha = snížení rizika chaotického postupu

- **Řízené plánem** (sledovat plán – hra na jistotu)
  - Vodopád, V-model
  - Kontext neměnný, zadání a technologie zřejmé, rozsah malý
- **Řízené riziky** (omezovat rizika)
  - Průzkumník/prototypování, Spirálový model
  - Kontext zřejmý, zadání a/nebo technologie nejasné
- **Řízené změnou** (adaptace na změnu)
  - Iterativní – RUP, agilní – SCRUM
  - Kontext proměnlivý, zadání a/nebo technologie nejasné

Základní modely životního cyklu

- **Sekvenční** - *velký třesk* (vztažené na celý produkt, naplánované pro celý projekt, oddělené meziprodukty)
  - *Vodopádový model*
- **Cyklický** - opakování technických aktivit, přírůstky (roste znalost, funkcionalita, kvalita, …)
  - *Spirálový model*
  - 

  - *RUP*
  - 

  - *Inkrementální (přírůstek je nějaká část (funkcionalita) produktu, produkt je funkční až na konci vývoje)*
  - *Iterativní (na konci každé iterace je funkční produkt, např. Na počátku jen se základní funkcionalitou, ale v dalších iteracích se funkcionalita postupně rozšiřuje)*
- **Agilní** - evoluce
  - extrémní programování *XP*

Základní modely životního cyklu software, softwarový proces, metodika.

Metodika

Konkrétní proces, má stanovené fáze, milníky, artefakty…

Základní modely životního cyklu software

Životní cyklus = proces od zahájení vývoje až po vyřazení z provozu

Sekvenční -> velký třesk

- Waterfall = vodopádový model, fáze jdou postupně od inception, elaboration, implementation, transition. Na konci je tzv. velký třesk, předání zákazníkovi – může dopadnout špatně, výsledek se nemusí shodovat s tím, co zákazník chtěl

Cyklický -> přírůstky/evoluce

- Spirálový model

Agilní -> Evoluce

Iterativní a inkrementální

- Pracuje se postupně v iteracích, na konci každé iterace je funkční produkt, jednotlivé iterace jsou jako samostatné malé projekty, které sestávají z několika fází

RUP – rational unified proces – od IBM, je to Framework, složitá implementace, častou chybou je, že se společnost rozhodne implementovat ten Framework celý – to je ale šatně, společnost by si měla vybrat jen to, co se jí hodí

OpenUP – částečně postaven na RUP, bere si z něj ale jen to nejnutnější pro vývoj software, je open source, distribuce formou wiki

SCRUM – daily standups, sprinty (30 dní), iterativní a inkrementální, plánuje jen následující iteraci, není předem jisté kdy projekt skončí

Softwarový proces

Softwarový process je systematická série aktivit, artefaktů a souvisejících rolí vedoucí k vyšší pravděpodobnosti úspěšného dokončení výroby software.

Výsledek = kvalitní software

Aktivity (Technické – modelování, programování, nasazení…, Komunikační – analýza, specifikace, …)

Artefakty (Dokument specifikace, Test protocol…)

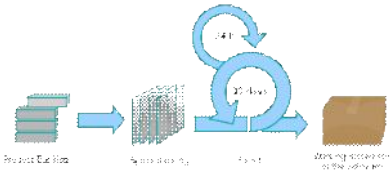
Role (Technické = analytik, team leader, developer, architect…, Manažerské = CEO, CIO…, Podpůrné = poradce, lektor…)

Řízené plánem – RUP

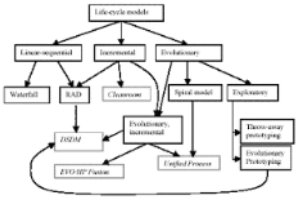
Řízené riziky – prototypování

Řízené změnou – SCRUM

- *SCRUM*



- Důraz na manifest:
  - Jednotlivci a interakce před procesy a nástroji
  - Fungující software před vyčerpávající dokumentací
  - Spolupráce se zákazníkem před vyjednáváním o smlouvě
  - Reagování na změny před dodržováním plánu



Agile vs Iterative vs Spiral

Agile Methods are a family of techniques that includes iterative development. Spiral is a type of iterative method that extends Waterfall methods. Spiral tends to be associated with longer iterations and normally involves a full development life cycle including extensive analysis and documentation.
The modern concept of Agile developed from the Agile Manifesto. Various practitioners from different software disciplines came together to articulate a common set of principles. These ideas incorporated concepts developed from extreme programming, unified process, scrum, test driven development, iterative programming, pair programming, continuous build, and others. To some degree, all of these so-called "practices" assist in the implementation of an Agile method. Every Agile project is somewhat different based on the composition of the team and the familiarity with the various practices used to build software.
Iterative development is a concept that can be applied across the continuum of methodologies ranging from extreme Agile to extreme Waterfall. Iterative development is characterized by the completion of a working system within a subset of time for the whole project and the repeated refinement of the working system based on cycles of development. In extreme programming iterations tend to be short, i.e. one to three weeks. Scrum based development methods advocate longer cycles of 2 to 6 weeks. Spiral development methods may have much longer iterations lasting several months.
Whether or not an iteration can be categorized as "agile" or "spiral" depends on the components of the iteration. The extreme agile iteration has little or no documentation, has many tests, and has a quick cycle. It may focus on one area of the overall system. A spiral iteration generally incorporates a complete development life cycle including analysis, planning, implementation, testing, and delivery.

From <http://agilemanagementresearch.blogspot.cz/2009/10/agile-vs-iterative-vs-spiral.html>

From <http://testconsultant.blogspot.cz/2008/08/development-models-and-impact-on.html>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> #

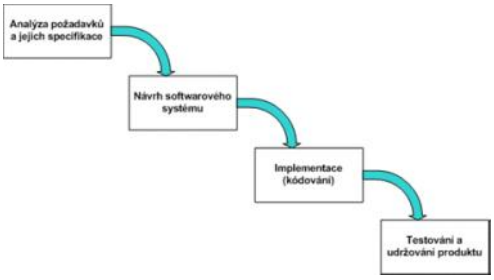
Sekvenční a iterativní přístup k vývoji software, výhody, nevýhody, důsledky, způsoby dodávky produktu.

Wednesday, May 29, 2013 4:50 PM

Sekvenční přístup

- vztažené na **celý** produkt- *velký třesk*
- naplánované pro celý projekt, oddělené meziprodukty
- vhodné pouze pro malé projekty s malou mírou neznáma (change management, náklady na změny, ...)
- předání až na konci celého projektu

- **Vodopádový model**
  - Životní cyklus projektu je rozdělen na 4 základní fáze:
    - Analýza požadavků a jejich specifikace
    - Návrh softwarového systému
    - Implementace (kódování)
    - Testování a udržování vytvořeného produktu
  - Následující množina činností spjatá s danou fází nemůže započít dříve než skončí předchozí



Výhody:

- snadné k pochopení
- dobrá možnost řízení a sledování postupu řešení (milníky (milestones) – voleny po činnostech)
- klade důraz na dokumentaci - specifikace, design, analýza

Nevýhody:

- vyžaduje mít na počátku přesně a úplně definované požadavky (uživatel často nedokáže stanovit předem)
- provozuschopnost verze vidí zákazník až v závěrečných fázích řešení, případné závažné nedostatky jsou odhaleny velmi pozdě.
- během vývoje se mohou měnit požadavky a výsledkem je, že dodaný produkt není to, co zákazník chtěl
- během implementace se zjistí, že design není v pořádku a je třeba ho změnit

Iterativní přístup

- Vývoj rozdělen na malé části, miniaturní úplné projekty s cca vodopádovým modelem, tzv. Iterace. (Charakteristika a vlastnosti / průběh iterace viz. Otázka 1.1.4.)
- Inkrementální rozšiřování produktu z původní hrubé formy do výsledné podoby -> Umožňuje postupné upřesňování požadavků na cílový produkt
- Řízen riziky a priority uživatele na funkčnost produktu
- Zaměření na architekturu produktu
- Požadavky mají zásadní vliv na návrh a implementaci produktu
- Řízení a plánování iterativně vedeného softwarového projektu viz. Otázka 1.1.5.

Výhody:

- Menší časové úseky v dodávkách produktu -> zvýšení úspěšnosti produktu díky zpětné vazbě
- Snížení rizik
- Snazší řízení změn na základě zpětné vazby uživatele
- Vyšší míra znovuvyužitelnosti
- Projektový tým se může učit během procesu vývoje
- Vyšší kvalita produktu

Nevýhody:

- Průběžné změny mohou způsobit porušení původní systémové struktury což vede k náročnější údržbě softwaru
- Vyžaduje přísnější management

Způsoby dodávky produktu

Velký třesk

- malé projekty, jasné požadavky

Přírůstkově

- určení přírůstků -> plán -> postupné dodávky
- zpětná vazba, ale úpravy projektu obtížné

Evolučně

- cyklus: určení cíle -> dodávka -> zpřesnění ("growing sw")

Sekvenční a iterativní přístup k vývoji software, výhody, nevýhody, důsledky, způsoby dodávky produktu.

Sekvenční a iterativní přístup k vývoji software

Sekvenční

- Skládá se z několika po sobě jdoucích fází, typický příklad takového procesu je waterfall – inception, elaboration, construction, transition
- Výhody:
  - Jednoduché na pochopení a implementaci
  - Vhodné pro velmi malé projekty s jednoznačným zadáním
- Nevýhody:
  - U větších projektů riziko, že výsledek nebude v souladu s tím, co si zákazník představoval
  - **Nutné přesně definovat požadavky**

Iterativní:

- Iterativní přístup rozdělí celý proces vývoje software na menší celky, tzv. iterace, kdy v rámci dané iterace probíhá vývoj sekvenčně a na konci je třeba mít funkční výstup.
- Výhody:
  - Na konci každé iterace je funkční výstup
  - Konzultace se zákazníkem, odchýlení se od požadavků se projeví včas a ne až na konci
  - Zákazník vidí progress
  - **Postupné zpřesňování požadavků**
- Nevýhody:
  - Složitější na implementaci než vodopád
  - Náročnější údržba software díky změnám v požadavcích

Způsoby dodávky produktu

Velký třesk

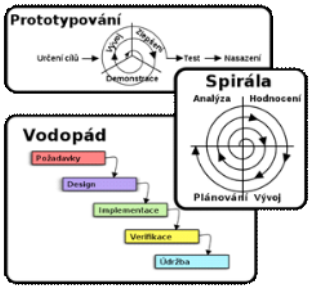
- Typicky u vodopádu, zákazník netuší až do konce jak produkt vypadá

Přírůstkově

- určení přírůstků, plán, úpravy složité

Evolučně

- Postupně dodáváme software ve funkčním stavu a přidáváme funkcionalitu do nových verzí – určení cíle, dodávka, zpřesnění

- 

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> w

Základní charakteristiky iterativních a agilních metodik.

Wednesday, May 29, 2013 4:51 PM

[aswi 01b-iterativni.pdf, 02a-zahajeni.pdf]

Agile Manifest

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

From <http://www.agilemanifesto.org/>

Průběh iterace

1) Plánování cíle iterace (funkčnost)
2) Doplnění a zpřesnění požadavků (základ: plán projektu, vize, předchozí feedback)
3) Dotváření návrhu
4) Implementace přírůstku funkčností
5) Integrace přírůstku (ověření, otestování)
6) Předání do provozu (validace zákazníkem)
7) Zhodnocení

Počet a pravidla iterací

Počet: charakter projektu, fáze vývoje, obvykle alespoň 3 celkem

Pevné datum ukončení: plánováno nejpozději na začátku iterace

Běžící iterace uzavřena změnám zvenčí (nutné pro stabilitu projektu), potřebuje dobré změnové a projektové řízení, zdroje tlaku na změnu: čas, funkčnost, postup

Délka iterace

Malá je lepší – blízký cíl, menší složitost/riziko, rychlá adaptace, vysoká produktivita

- 1-4 týdny pro malé, 3-6 týdnů velké projekty, zřídka měsíce

Vždy pevné datum ukončení

Timeboxované iterace = délka známa předem

Milníky

- **LCO** (Lifecycle Objectives)
  - Srozumění s rozsahem, cenou, harmonogramem
  - Souhlas s požadavky a jejich klíčovostí
  - Navrhovaný postup vývoje souhlasí
  - Rizika identifikována a řešení známo
  - **Artefakty**
    - Vize produktu, business case
    - Seznam rizik a strategie jejich řešení
    - Slovník pojmů a přehled klíčových požadavků
    - Koncept technického řešení (architektura + prototypy)
    - Plán projektu
    - Popis procesu a infrastruktury
- **LCA** (Lifecycle Architecture)
  - Vize a klíčové požadavky jsou stabilní
  - Testy ověřily, ze architektura řeší rizikové požadavky/faktory
  - Jsou přesnější odhady pracnosti, na nich postaveny plány
  - Nástroje a postupy pro realizaci jsou v provozu
  - Stakeholders: vize realizovatelná, spotřebované zdroje adekvátní
  - **Artefakty**
    - Vize produktu
    - Seznam rizik a strategie jejich řešení
    - Popis architektury, validační testy
    - Plán projektu, popis infrastruktury
    - Dokument specifikace požadavků (DSP)
- **IOC** (Initial Operational Capability)
  - Je hotová beta verze produktu
  - Je hotová první verze plánu nasazení
  - Implementace je dokumentovaná, existují používané testy
  - Je rozpracována uživatelská dokumentace
  - Jsou aktualizovány popisy návrhu, datového modelu, požadavků
  - **Artefakty**
    - Plán nasazení
    - Testovací sady + reporty
    - Architektura (aktualizována) + popis implementace
    - Uživatelská příručka
- **GA** (General Availability)
  - Uživatel je spokojen s produktem
  - Stakeholders jsou spokojeni s produktem
  - uvést produkt do rutinního provozu – „krabice“ s produktem, website launch, raut :-)
  - support team v provozu
  - **Artefakty**
    - Release produktu
    - Podpůrné materiály (uživatelská dokumentace)
    - Baseline kompletní konfigurace release

Charakter iterací dle fáze

Základní schéma pevné, mění se činnosti a artefakty

1) Zahájení – analytické činnosti, validace vize zákazníkem (1-2 iterace)
2) Projektování – analytické a designérské činnosti, ověřování prototypy, implementace (2+ iterací)
3) Konstrukce – designérské a programátorské činnosti, změnové řízení, testování a ověřování (N iterací)
4) Nasazení – integrační a konzultační činnosti, ověřování provozem, náběh uživatelské podpory (1-2 iterace)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Základní charakteristiky iterativních a agilních metodik.

Iterativní a agilní metodiky

Postupujeme po jednotlivých iteracích, v rámci jedné iterace postupujeme sekvenčně (ala vodopád), na konci každé iterace musí být k dispozici funkční software.

V rámci iterace máme fáze: requirements, design, implementation, testing, deploy

V rámci všech iterací jsou to pak fáze: inception, elaboration, construction, transition

Funkční software můžeme předvést zákazníkovi a na základě zpětné vazby upřesnit požadavky.

Konce iterace jsou pevně dány, pokud se nestíhá, je možné omezit funkcionalitu, ale konec iterace nelze posunout. Během iterace nemůže dojít ke změně požadavků, je to zakázáno. Na konci iterace se hned plánuje další iterace.

Minimum jsou alespoň 3 iterace, ale může jich být samozřejmě více, záleží na charakteru projektu. Délka iterací je typicky 2 týdny, ale může být více i méně. Na konci každé iterace je provedeno review, neboli zhodnocení.

Milníky

LCO

Life Cycle Objectives – milník, kdy již máme první verzi specifikace požadavků, seznam rizik, cena, rozsah, harmonogram…

Artefakty: vize produktu, business case, seznam rizik a strategie řešení, slovník pojmů a přehled klíčových požadavků, plán projektu, koncept technického řešení (architektura + prototypy)…

LCA

Life Cycle Architecture – milník, kdy již máme ustálenou architekturu systému, zpřesněnou specifikaci požadavků, ověřenou architekturu, eliminována kritická rizika (prototypy)

Artefakty: zpřesněná vize, seznam rizik a jejich řešení, popis architektury, validační testy…

IOC

Initial Operational Capability – milník, kdy máme funkční spustitelnou verzi programu, která už vykonává základní funkcionalitu (beta)

Artefakty: spustitelná verze aplikace, dokumentace, uživatelský manuál, testovací sady + reporty, plán nasazení…

GA

General Availability – milník, kdy support tým je v pohotovosti, software v krabicích připraven k distribuci a my organizujeme raut

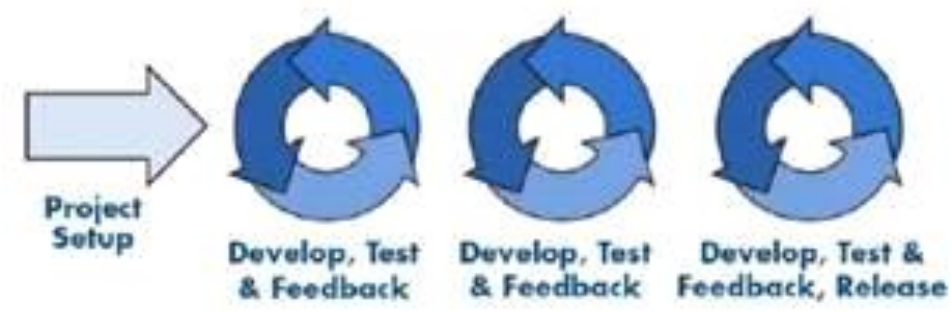
Artefakty: Release produktu, podpůrné materiály, baseline kompletní konfigurace

Vlastnosti iterace, její průběh.

Wednesday, May 29, 2013 4:52 PM

Vlastnosti iterace

- miniaturní úplný projekt s cca vodopádovým modelem, prolínání aktivit.
- cílem je iterační release (otestovaný, funkční ale funkčně neúplný produkt).
- cyklické opakování: Develop, Test, Feedback (pro celý projekt viz obr. níže)



- min 3 iterace na projekt (závisí na projektu, velikosti týmu,…)
- datum konce iterace se volí vždy na začátku – *timeboxované iterace* (XP 2 týdny, SCRUM 30 dní)
- iterace je uzavřená změnám zvenčí (pro stabilitu projektu)
- když se nestíhá je možné omezení plánované funkčnosti, ale ne přesčasy, nehotový release, měnit datum
- předávání po částech (konce iterace) - artefakty, demo, retrospektiva (!)
- každá iterace končí vytvořením spustitelného kódu

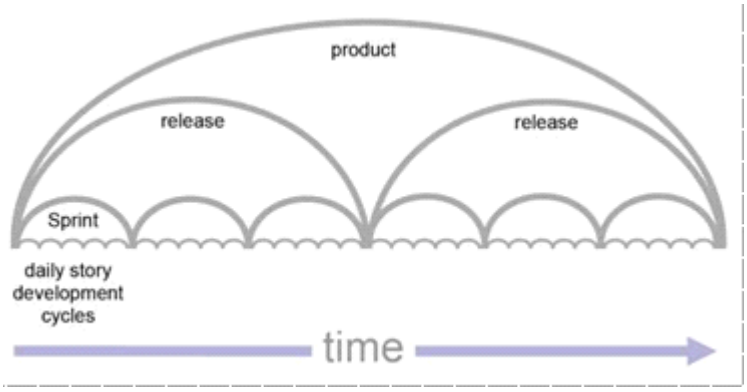
Průběh iterace

- 

Plánování cíle iterace (funkčnost)

- Doplnění / zpřesnění požadavků
  - Základ: plán projektu, vize, předchozí feedback
- Dotváření návrhu
- Implementace přírůstku funkčnosti
- Integrace přírůstku
  - Ověření, otestování
- Předání do provozu
  - Validace zákazníkem
- Zhodnocení

Kontext iterace v procesu vývoje



From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Plánování a řízení iterativně vedeného softwarového projektu.

Wednesday, May 29, 2013 4:52 PM

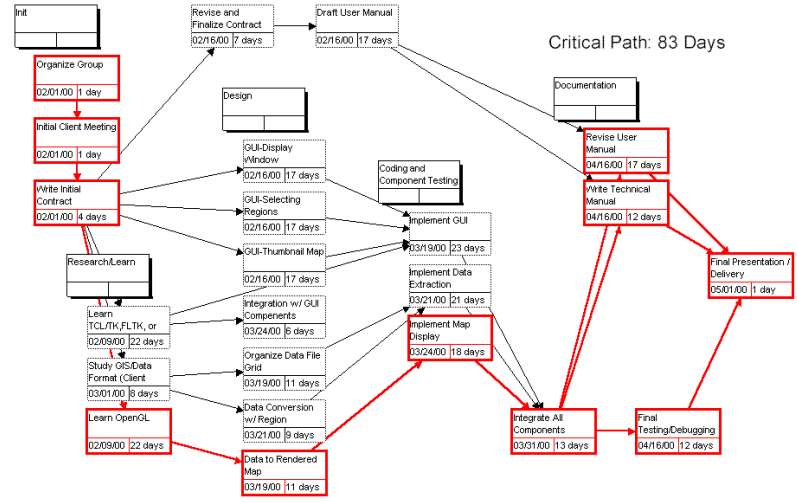
Plánování

Základní aspekty plánování

- Nějaký plán je nutný vždy
  - Harmonogram (termíny)
  - Pevné body
  - Přiřazení zdrojů
- Sledování plánu nutné vždy
  - Kontrola postupu
  - Reakce na změny
- Způsoby plánování
  - Prediktivní / adaptivní
  - Rizika / priority / ROI

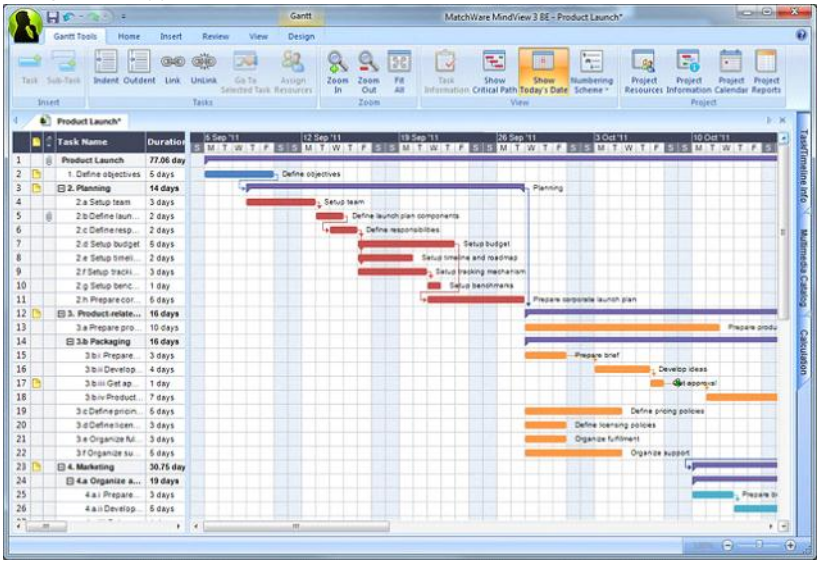
Obecně

- Typické pro sekvenční postup
- WBS (Work Breakdown Structure) -> PERT (Program Evaluation and Requirements Technique) -> Gantt
- PERT



From <http://www.cs.unc.edu/~stotts/145/homes/map/images/pert.gif>

- PERT - graf činností a jejich závislostí, hledá se kritická cesta



From <http://www.bing.com/images/search?q=gantt&view=detail&id=785B688245226CB96D33575FE691D434D498DFBC&FORM=IDFRIR>

- Gantt - graf, který ukazuje čas potřebný k dokončení projektu
- Pevné body v plánu
- Základní problém = velká míra nejistoty
  - Neznalost odhadů v době, kdy jsou potřeba
  - Měnící se požadavky -> rozsah projektu

Adaptivní plánování

- **Základní přístup**
  - Detailně plánovat možné jen to, na co máme data
  - Přesnější odhady a plán až po nějaké době
  - Plán na N+1 krok zpřesněn (adaptován) poznatky z N
- Globální (pevné) body v plánu dány/určeny předem
- **Stupně volnosti při plánování**
- Klasicky: čas, zdroje (cena), kvalita

Plánování a řízení iterativně vedeného softwarového projektu.

Plánování

- Nějaký harmonogram projektu je nutný vždy, má pevné body, je nutné rozdělit zdroje…
- Je nutné sledovat plán a reagovat na nečekané události

Provádí se na:

- Začátku projektu – rozvržení celého projektu, hlavní cíle, odhady
- Na začátku iterace – co se bude v dané iteraci provádět
- V průběhu iterace se pokud možno neplánuje vůbec

Obecně: WBS -> PERT -> Gantt

Adaptivní plánování – detailně plánovat je možné jen to, na co máme data – tzn. V kroce N můžu plánovat jen na N+1.

Stupně volnosti při plánování jsou 4: čas, zdroje, kvalita, funkčnost. Nejlépe se krátí ta funkčnost, ostatní faktory jsou víceméně neměnné.

Plánování v iteracích

- Plán projektu – vize, milníky
- Iterace
  - Plánovací schůzka
    - Výběr funkčnosti, odhadování, commitment
  - Sledování průběhu
    - Burndown chart, případné přeplánování pokud se nestíhá
  - Retrospektiva
    - Hodnocení, úpravy procesu, project velocity
- Plánovací schůzka - sledování průběhu - retrospektiva

Řízení

Iterace – plánování na začátku iterace, mají fixní deadline.

Fáze vývoje: inception, elaboration, construction, transition

Jednotlivé fáze iterace: plánování, návrh, implementace, testování, nasazení

Řízení je možné dvěma způsoby:

- Řízení riziky
  - Snažíme se nejprve eliminovat největší rizika, proto je plánujeme co nejdříve
- Řízení zákazníkem
  - Necháme na zákaznícovi, aby nám řekl, co potřebuje jako první – jsme ale omezeni délkou iterace, tzn. Můžeme udělat jen to, co za tu iteraci stihneme

Při řízení je důležitý backlog, na základě kterého se nám generuje burndowsn chart, který nám říká, jak na tom jsme – jestli práce stagnuje, přibývá, nebo uspokojivě ubývá.

    - Obtížně měnitelné, odhadovatelné
    - Kvalita obtížně řiditelná
    - Agilně: + funkčnost
        - Nejlepší faktor pro řízení projektu
        - Funkčnost je totiž na rozdíl od předchozích nejsnáze měnitelná
        - Vhodná granularita -> snadné a přesné odhady
- **Plánování podle rizik / priorit**
    - Řízení riziky (rizika jsou přímá a nepřímá - ty nepřímá téměř nelze ovládat)
        - Vyhodnotit rizikové faktory projektu
        - Začít částmi funkčnosti/designu s největší mírou rizika
    - Řízení prioritami klienta
        - Výběr funkčnosti je na zákazníkovi
        - Množství funkcí je omezeno délkou iterace
        - Umožňuje pružně reagovat na aktuální potřeby
- **Okamžiky plánování**
    - Na počátku projektu
        - Hlavní cíle, hrubé odhady
    - Na začátku každé iterace
        - Seznam aktivit (úkolů)
        - Odhadování pracnosti -> časů, zdrojů
        - Vyřazení dle priorit (rizika, klient)
    - V průběhu iterace
        - … se pokud možno neplánuje
- **Dokumentace plánování**
    - Podklady
        - Rozpad prací WBS
        - Zdroje a jejich alokace
    - Grafické nástroje
        - PERT graf - návaznosti, kritická cesta
        - Gantt chart - čas, zdroje
    - Dokumenty
        - Vize a rozsah projektu/nabídka
        - Plán projektu
        - Plán pro řízení rizik
        - Plán iterace

Plánování a řízení iterací

- Iterace je miniaturní projekt
- Cíl = podmnožina požadavků na kompletní produkt
- Zachyceno v plánu iterace
    - Backlog
    - Bugtracker
    - Dokument

Postupy plánování

- Direktivní
- Týmové
- Planning Game

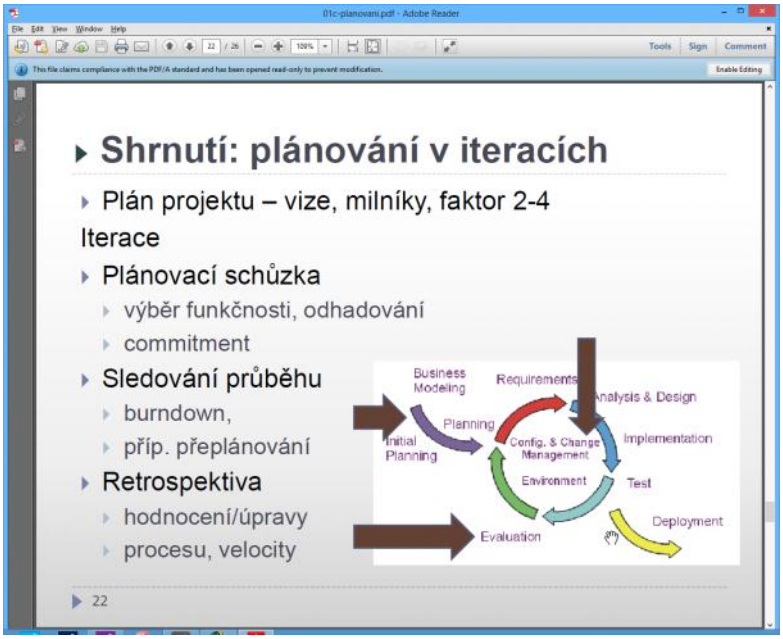
> **Pieces:** The basic playing piece is the UserStory. Each Story is written on an IndexCard. Stories have a value and a cost, although this is a little tricky because the value of some Stories depends on the presence or absence of other Stories (see StoryDependenciesInXp), and the values and costs change over time.
> **Goal:** The goal of the game is to put the greatest possible value of stories into production over the life of the game.
> **Players:** The players are Business and Development.
>
> From <http://c2.com/cgi/wiki?PlanningGame>

Pomocné metody

- Dot voting
- Backlog grooming

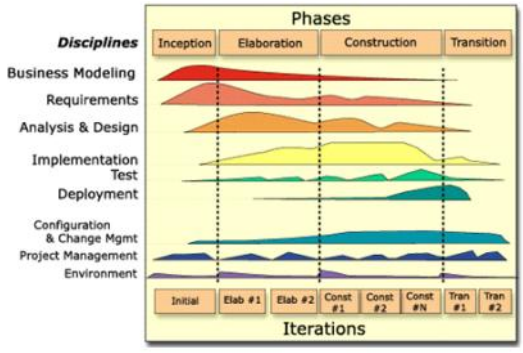
Pravidla

- Pevné datum konce, timebox
- Priority závazné



Globální řízení iterativního projektu

- Výchozí bod: **vize** produktu
- Oddělené sekvenční **fáze** reprezentující „klasické" inženýrské disciplíny. Každá fáze má jasné rozdělení cílů a výsledků. Skládá se z 1 až N iterací. Fáze:
  - **Zahájení** (1 – 2 iterace)
    - analytické činnosti, validace vize zákazníkem
  - **Projektování** (2+ iterace)
    - analytické a designérské činnosti, ověřování prototypy, implementace
  - **Konstrukce** (N iterací)
    - designérské a programátorské činnosti, změnové řízení, testování a ověřování
  - **Nasazení** (1 – 2 iterace)
    - Integrační a konzultační činnosti, ověřování provozem, náběh uživatelské podpory



Globální plánování:

Milníky voleny na základě stupních přesnosti (produktu) a míře rizika

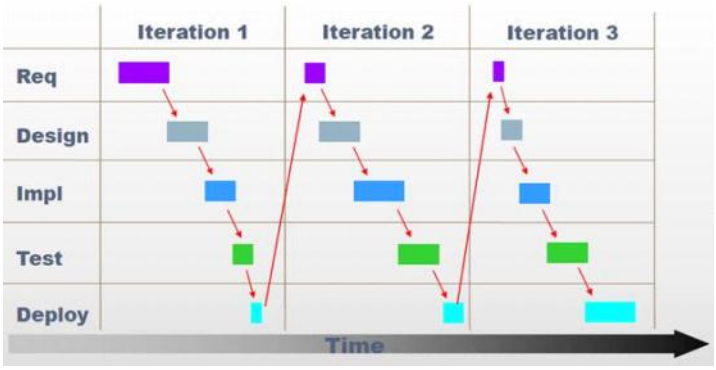
Milníky:

- **LCO** (Lifecycle Objectives)
  - Srozumění s rozsahem, cenou, harmonogramem
  - Souhlas s požadavky a jejich klíčovostí
  - Navrhovaný postup vývoje souhlasí
  - Rizika identifikována a řešení známo
  - **Artefakty**
    - Vize produktu, business case
    - Seznam rizik a strategie jejich řešení
    - Slovník pojmů a přehled klíčových požadavků
    - Koncept technického řešení (architektura + prototypy)
    - Plán projektu
    - Popis procesu a infrastruktury
- **LCA** (Lifecycle Architecture)
  - Vize a klíčové požadavky jsou stabilní
  - Testy ověřily, ze architektura řeší rizikové požadavky/faktory
  - Jsou přesnější odhady pracnosti, na nich postaveny plány
  - Nástroje a postupy pro realizaci jsou v provozu
  - Stakeholders: vize realizovatelná, spotřebované zdroje adekvátní
  - **Artefakty**
    - Vize produktu
    - Dokument specifikace požadavků
    - Seznam rizik a strategie jejich řešení
    - Popis architektury, validační testy
    - Plán projektu, popis infrastruktury
- **IOC** (Initial Operational Capability)
  - Je hotová beta verze produktu
  - Je hotová první verze plánu nasazení
  - Implementace je dokumentovaná, existují používané testy
  - Je rozpracována uživatelská dokumentace
  - Jsou aktualizovány popisy návrhu, datového modelu, požadavků

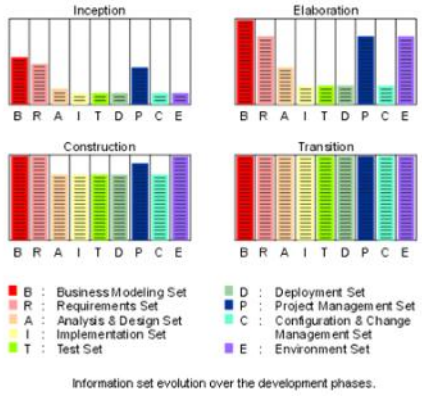
- **Artefakty**
  - Plán nasazení
  - Testovací sady + reporty
  - Architektura (aktualizována) + popis implementace
  - Uživatelská příručka

- **GA** (General Availability)
  - Uživatel je spokojen s produktem
  - Stakeholders jsou spokojeni s produktem
  - uvést produkt do rutinního provozu – „krabice“ s produktem, website launch, raut :-)
  - support team v provozu
  - **Artefakty**
    - Release produktu
    - Podpůrné materiály (uživatelská dokumentace)
    - Baseline kompletní konfigurace release

Charakter iterací dle fáze



Artefakty dle fáze:



Information set evolution over the development phases.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> A

Požadavky na software – typy požadavků, formy popisu, úrovně detailu a jejich vztah k procesu vývoje.

Wednesday, May 29, 2013 4:53 PM

Co je to požadavek?

- *požadavek* = schopnost nebo vlastnost, kterou má sw mít, aby jej uživatel mohl použít k vyřešení problému nebo dosažení cíle, který vedl k zadání, nebo aby splnil podmínky stanovené smlouvou, standardem nebo jinou specifikací.
- vlastnosti požadavku: úplný, bezesporný
- požadavkem není to, co uživatel nepotřebuje

Typy požadavků

- **Funkční požadavky = funkce**
  - popisují funkce nebo služby, které jsou od systému očekávány
  - příklady: požadavky na univerzitní knihovní systém
- **Mimofunkční požadavky = vlastnosti**
  - netýkají se funkcí systému, ale vlastností jako je spolehlivost, čas odpovědi, obsazené místo na disku nebo v paměti, aj.
  - často kritičtější než jednotlivé funkční požadavky (např. pokud je řídící systém letadla nespolehlivý, je nepoužitelný)
  - někdy dané vnějšími faktory, tj. legislativní požadavky (př. zákon na ochranu osobních údajů apod.)
  - př. veškerá komunikace mezi uživatelem a systémem by měla být vyjádřitelná ve znakové sadě ISO 8859-2
- **Business požadavky**
  - Vize a rozsah projektu
  - Smluvní záležitosti
  - Náklady, TCO
- **Právní a další**
  - Např. různé právní systémy dvou zemí a cloud

Způsob formulace požadavku

- uživatelská specifikace
- vysokoúrovňový popis funkčních a mimofunkčních požadavků zákazníka
- musí být srozumitelné pro uživatele, kteří nemají technické znalosti
- systémová specifikace
- podrobnější specifikace uživatelských požadavků pro vývojáře
- slouží jako výchozí bod pro design systému

Formy popisu

- textový popis
- shopping list
- strukturovaný text
- grafické vyjádření
- use case diagramy
- ERA, UML
- implementace
- popis ve formě prototypu a uživatelské příručky

Úrovně detailu v rámci procesu vývoje

- Zahájení projektu: strategické, klíčové, obrysy
- Projektování: podstatné, úplnost
- Konstrukce: podrobnosti

Úroveň detailu a agilní metodiky

- Cíl: zachytit věci v danou chvíli nejpodstatnější (viz Manifest), detaily se dohodnou až bude potřeba
  - Zákazník musí dát podklad pro odhad a plánování
- S každou iterací zpřesnění
  - User stories
  - Tasky v backlogu
  - Jednotlivé úkoly

Dokument specifikace požadavků (DSP)

- konečný výsledek analýzy požadavků
- kompletní popis chování systému
- zahrnuje případy užití popisující všechny interakce uživatele se SW -- funkční požadavky
- technický dokument, oficiální vyjádření o tom, co se od vyvíjeného systému očekává (dohoda mezi zákazníkem a dodavatelem, co má zadaný sw dělat a jak to má vypadat)
- základ pro pozdější ověření správnosti - důraz na jednoznačnost, ověřitelnost, reálnost, srozumitelnost, úplnost, přesnost a správnost, modifikovatelnost, konzistenci
- měl by specifikovat pouze externí chování systému, tj. snaha vyloučit design z DSP
- strukturován tak, aby v něm bylo snadné provádět změny (modifikovatelnost)
- měl by specifikovat omezení implementace – mimofunkční požadavky
- měl by charakterizovat přijatelné odpovědi na nežádoucí události

Jednodušeji:

Jednoznačnost, úplnost, srozumitelnost, modifikovatelnost, přesnost, ověřitelnost, reálnost, specifikace pouze chování - NE jak to udělat

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> mo

Požadavky na software – typy požadavků, formy popisu, úrovně detailu a jejich vztah k procesu vývoje.

Požadavek = schopnost nebo vlastnost, kterou by měl výsledný produkt mít

Požadavkem není to, co uživatel nepotřebuje

Typy požadavků:

Funkční požadavky

- Popisují funkce nebo služby, které jsou od systému očekávány

Mimofunkční požadavky

- Neméně důležité, např. požadavky na spolehlivost systému, rychlost odezvy, dané vnějšími faktory – legislativa…

Business požadavky

- Cena, vize, rozsah projektu, smluvní záležitosti…

Způsoby formulace požadavku

- Slovně, formou pár vět
- User stories
- Use case diagram
- Obrázek
- Shopping list
- Uživatelská příručka

Úrovně detailu:

- Na začátku jsou požadavky spíše obecné, časem s postupem projektu se zpřesňují
- Tasky v backlogu, jednotlivé úkoly

DSP

- Konečný výsledek analýzy požadavků
- Kompletní popis chování systému
- Zahrnuje případy užití zahrnující všechny případy interakce uživatele se software
- Pouze externí chování z pohledu uživatele
- Strukturován, snadné úpravy
- Technický dokument, oficiální vyjádření o tom, co se od produktu očekává

- Jednoznačnost, srozumitelnost, modifikovatelnost, snadné úpravy, formálnost – specifikuje CO udělat NE JAK to dělat

Postupy pro sběr požadavků.

Wednesday, May 29, 2013 4:54 PM

Problematická a ošemetná to záležitost:

- Uživatelé nerozumí tomu, co chtějí, nebo uživatelé nemají jasnou představu o svých požadavcích
- Uživatelé neschválí seznam sepsaných požadavku jako finální
- Uživatelé trvají na nových požadavcích i po zafixovaní nákladů a časového harmonogramu
- Komunikace s uživateli je pomalá
- Uživatelé se často nepodílejí na kontrolách, nebo jsou neschopní to udělat
- Uživatelé jsou technicky nevzdělaní
- Uživatelé nerozumí procesu vývoje
- Uživatelé neví o současné technologii
- Je potřeba předem počítat s různou úrovní počítačové gramotnosti. To může vést k situaci, kdy uživatelé průběžně mění své požadavky, i když systém nebo vývoj produktu byl zahájen.

Způsoby sběru požadavků

- **Neinteraktivní**
  - *analýza existujícího systému*
    - inspirujme se tím, jak funguje stávající systém
  - Studium dokumentace
  - Hlášení problémů
  - Analýza trhu
  - Konkurenční systémy
- **Interaktivní**
  - *interview*
    - předem připravený rozhovor, který vede moderátor (klade otázky, dává slovo)
    - nedoporučuje se více než 2 hodiny
    - předem si připravit scénář, které okruhy se budou probírat, v jakém pořadí, scénář se snažit nenásilně dodržovat
  - *pozorování, práce s uživateli*
    - pozorování prací u zákazníka (účast analytiků)
  - *dotazníky*
    - vhodnými otázkami zjistíme od uživatelů, co potřebují
  - *prototypování* – tvorba prototypů, podle kterých si zákazník ujasní své požadavky
    - stačí na papír nebo skutečné programové prototypy
  - *studium hlášení problémů*

Způsoby vyjádření

- *přirozený jazyk*
  - výhodou je srozumitelnost pro uživatele
  - nevýhodou – spoléhá se na to, že autoři používají stejná slova pro stejný koncept (stejná věc se dá říci mnoha různými způsoby). Obtížná modularizace - kterých všech dalších požadavků se změna dotkne.
- *formuláře*
  - pro vyjádření požadavku se nadefinuje jeden nebo více typů formulářů
  - měl by obsahovat:
    - popis specifikované funkce nebo entity
    - popis vstupů, odkud se berou
    - popis výstupů, kam putují
    - jaké další entity specifikovaná funkce nebo entita používá
    - případné pre/post conditions (co platí při vstupu do funkce a co při výstupu z ní)
    - pokud vznikají postranní efekty, pak jejich popis
- *pseudokódy*
  - v přirozeném jazyce těžko vyjádřitelné vnořené podmínky nebo smyčky
  - jazyk s abstraktními konstrukcemi, které právě potřebujeme
  - vnoření konstrukcí je vyjádřeno odsazením
  - vyhýbáme se syntaktickým konstrukcím cílového programovacího jazyka (popisujeme požadovaný záměr, nikoli jak to bude v cílovém jazyce)
  - na druhou stranu musí umožňovat téměř automatickou konverzi do kódu
- *Obrázky, protoyp GUI*

Kontrola požadavků

- musíme zjistit, zda jsou požadavky úplné, konzistentní a zda odpovídají tomu, co zadavatel chce
- vstupem je úplný **Dokument specifikace požadavků**
- metody:
- přezkoumání (reviews) – požadavky jsou systematicky kontrolovány týmem, manuální proces
- prototypování – zákazníkovi předvedeme spustitelný model systému
- generování testovacích případů – vytvoříme testy požadavků, pokud je obtížné vytvořit test, bude požadavek obtížně implementovatelný
- automatická analýza konzistence – pokud byly požadavky specifikovány jako model ve formální nebo strukturované notaci

Management požadavků

- požadavky na systém se stále mění
- měl by začít plánováním, ve kterém se rozhodne:
  - **způsob identifikace požadavků** – každý požadavek by měl mít jednoznačné ID
  - **proces změny požadavků** – definujeme proces, abychom se ke změnám požadavků chovali konzistentním způsobem
  - **sledovatelnost**
    - zdroj požadavku – kdo požadavek navrhnul, důvod; abychom se mohli zdroje zeptat na podrobnosti
    - vztahy mezi požadavky – kolika požadavků se změna dotkne
  - **nástroje** – co se použije pro uchování informací o požadavcích (malé projekty – obvyklé prostředky(textové nástroje, EXCEL, databáze, aj), velké projekty – CASE nástroje)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Postupy pro sběr požadavků.

- Uživatel často neví, co chce
- Požadavky upřesňuje postupně, takže se v počáteční fázi musíme snažit co nejlépe zjistit co přesně se od nás očekává
- Sběr požadavků většinou zajišťuje analytik, který funguje jako styčný bod mezi zákazníkem a týmem vývojářů

Interaktivní

- Interview – rozhovor se zákazníkem
- Pozorování zákazníka – sledování chování zákazníka za účelem zjištění co nejvíce informací o procesu
- Prototypování – na základě prototypů se ukáže zákazníkovi, jak by výsledný produkt mohl vypadat
- dotazníky

neinteraktivní

- analýza existujícího systému
- inspirace konkurencí
- studium dokumentace
- analýza trhu

Způsoby vyjádření požadavků

- pseudokódy
- přirozený jazyk
- formuláře
- prototypy

Měli bychom začít tím, že rozhodneme, jak budeme požadavky identifikovat, jak je budemem případně měnit, jak vysledujeme, kdo požadavek navrhnul, na čem požadavek závisí apod. K tomu slouží nástroje – v nejjednodušším případě nám postačí Excel, jinak je lepší použít nějaký CASE systém.

Analýza požadavků a tvorba objektového návrhu – postup, použité modely a diagramy.

Wednesday, May 29, 2013 4:55 PM

Analýza požadavků obsahuje tři typy aktivit:

- **Sběr požadavků:** komunikace se zákazníky a uživateli za účelem získání jejich požadavků na systém.
- **Analýza požadavků:** identifikování nejasných požadavků, nekompletních, nejasných, nebo protichůdných a následně řešení těchto nesrovnalostí.
- **Zaznamenání požadavků:** dokumentování požadavků v různých formách, jako běžný textový dokument, případy užití (use case), nebo specifikace procesů → dokument specifikace požadavků

Analýza a klasifikace požadavků

- Identifikace zúčastněných stran ("Stakeholderů")
- Případy užití (Use Case)
- XP (Extrémní programování) – user stories

- **Funkční**
- **Mimofunkční**
  - Výkonnostní
  - Designové
  - Zákonný rámec

Lidé v analýze

- **Zákazník**
  - Externí, interní
  - Doménový expert
- **Zainteresovaný hráč**
  - Ředitel, investor, standardizační orgán, daňový poplatník…
  - Vliv na úspěch projektu
- **Analytik**
  - Hlavní úloha
    - Zprostředkovatel mezi zákazníkem a programátory
  - Dovednosti a vlastnosti
    - Komunikační schopnosti, naslouchání a pozorování
    - Vedení schůzek, organizační
    - Schopnost abstrakce, nadhled, tvořivost
    - Detailní znalost problémové oblasti
    - Psaný projev, modelování
- Requirements development
  - Elicit
  - Analyze, negotiate
    - Potential -> stable requirements
  - Document
  - Review
    - Baselined requirements
- Requirements management
  - Change management

Metoda FURPS – model klasifikace funkčních a mimofunkčních požadavků

- F (functionality) – funkčnost
- U (usability) – užitečnost
- R (reliability) – spolehlivost
- P (performace) – výkon
- S (supportability) – rozšiřitelnost

Kontextový model

- Zobrazení vztahů systému s okolními entitami
  - Systém jako černá skříňka
  - Aktéři, stakeholders
  - Ostatní systémy
- Rozsah systému
- Rozhraní na okolí
  - HCI, API, data

4(+1) kategorie požadavků

- Klasifikace funkčnosti do 4 kategorií
  - Jaké informace systém obsluhuje, udržuje
  - Jaké funkce poskytuje uživatelům
  - Jaké analýzy dat provádí
  - Jaké jsou interakce s jinými systémy
- Pro každý druh příslušné vlastnosti
  - Stačí jednoduché seznamy
- +1 = mimo rozsah zadání, doplňková funkčnost

Vize produktu a rozsah projektu

- Co má vzniknout
  - Esenciální, klíčové, vysokoúrovňové požadavky
  - Problem behind the problem
  - Business case (proč se to tak chce)
    - Zdůvodnění výhodnosti-návratnosti projektu
  - Požadavky: co to má dělat
- Popis problému
  - Obchodní příležitost
- Přehled stakeholders a uživatelů
  - Vize zahrnuje pohled všech zainteresovaných účastníků
  - Kdo jsou zájemci o systém, potenciální konkurence
- Přehled očekávaných schopností a funkcí produktu

Analýza požadavků a tvorba objektového návrhu – postup, použité modely a diagramy.

Typu požadavků:

  - Funkční
  - Mimofunkční

Lidé v jejich analýze:

- Zákazník
  - Může být externí nebo interní
  - Doménový expert
- Zainteresovaný hráč
  - Ředitel, investor, …
  - Má vliv na úspěch projektu
- Analytik
  - Jeho hlavní úlohou je zprostředkovávat kontakt mezi programátory a zákazníky
  - Musí mít dobré komunikační schopnosti, schopnost abstrakce, detailní znalost problémové oblasti, organizační dovednosti

Postup sběru požadavků

- **Elicit**
- **Analyze, negotiate**
- **Document**
- **Review**

Kategorie požadavků

- Podle toho, jaké informace systém obsluhuje, udržuje
- Jaké funkce poskytuje uživatelům
- Jaké analýzy dat provádí
- Jaké má interakce s jinými systémy

Kontextový model

- Zobrazení vztahů s okolními entitami
- Systém jako černá skříňka
- Zobrazuje vztahy se stakeholders + ostatními systémy včetně rozhraní

Vize produktu

- Co má výsledný produkt dělat
- Kdo jsou stakeholders
- Kdo jsou uživatelé
- Omezení
- Časový rámec plánu projektu

Model užití

- Model = aktéři + případy užití
- Případ užití = sekvence akcí, které systém provede v důsledku nějakého podnětu zvenčí a jejichž výsledek je viditelný pro uživatele

Doménový model

- Pro modelování datové části
- Musíme od klienta zjistit, co tyto doménové třídy budou
- S těmito třídami budeme najisto počítat
- Doménový model -> logický datový model

Datový model

- Logický (konceptuální), fyzický – obvykle ERA, viz DB1
- Hodnoty atributů datových entit se vyvíjí v čase

CRUDL matice – Create, Rename, Update, Delete List – cílem je vědět kdo a jak manipuluje s údaji

Modely základních struktur systému

Glosář

- Pojmy a jejich vysvětlení, prevence nedorozumění

Doménový model

- Model základních entit a jejich vztahů, doménové objekty – diagram tříd

Model nasazení

- Vztah produktu k prostředí

    - Popis, kvalitativní charakteristiky, priority
- Omezení, standardy, závislosti vztahující se k projektu
- Rámec plánu projektu
    - Časový rozsah

Model užití

- Popis požadavků na vnější funkčnost systému
    - Jací uživatelé k systému přistupují
    - Co to pro ně dělá
    - Jak systém zpracovává požadavky
    - Kde je hranice systému
- Model = aktéři + případy užití
    - Kontext, primární funkčnosti
    - Další iterace, podružná funkčnost
    - #### Případy užití
        - Co to je případ užití
            - Sekvence akcí, které systém provádí v důsledku nějakého vnějšího podnětu a které vedou k výsledkům viditelným pro některého jeho uživatele
        - Aktéři …
        - Hledáme dialogy typu aktér-systém
        - Základní popis případu užití: název, stručný popis účelu, základní kroky postupu, odkazy na zdrojové informace
        - UML diagram případů užití

Tvorba objektového návrhu

Diagram aktivit

Diagram aktivit (UML)

- **akce** – atomické dále nedělitelné kroky
- **vnořené aktivity** – volání jiných procesů (aktivit), tyto aktivity mohou být reprezentovány dalším **diagramem aktivit**. Sekvenci jednotlivých kroků v diagramu aktivit určuje řídicí tok.

Modelování datové části

Použití doménového modelu

- Komunikace
    - Dorozumění s klientem
- Objektová analýza a návrh
    - S těmito třídami můžeme najisto počítat
    - Analýza přidá zodpovědnosti, detaily vlastností a chování
    - Návrh je transformuje a přidá implementační třídy
- Doménový model -> logický datový model

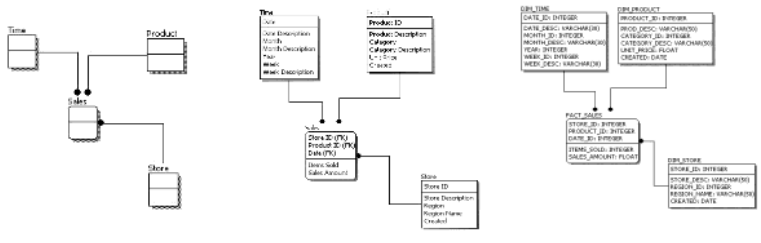
Datový model

The three level of data modeling, conceptual data model, logical data model, and physical data model, were discussed in prior sections. Here we compare these three types of data models. The table below compares the different features:

Feature	Conceptual	Logical	Physical
Entity Names	✓	✓	
Entity Relationships	✓	✓	
Attributes		✓	
Primary Keys		✓	✓
Foreign Keys		✓	✓
Table Names			✓
Column Names			✓
Column Data Types			✓

Below we show the conceptual, logical, and physical versions of a single data model.

Conceptual Model Design **Logical Model Design** **Physical Model Design**



We can see that the complexity increases from conceptual to logical to physical. This is why we always first start with the conceptual data model (so we understand at high level what are the different entities in our data and how they relate to one another), then move on to the logical data model (so we understand the details of our data without worrying about how they will actually implemented), and finally the physical data model (so we know exactly how to implement our data model in the database of choice). In a data warehousing project, sometimes the conceptual data model and the logical data model are considered as a single deliverable.

From <http://www.1keydata.com/datawarehousing/data-modeling-levels.html>

- Logický (konceptuální), fyzický
  - Obvykle jeden z prvních nezávisle na OO technikách (viz db1/db2/psds)
- Vývoj datových entit
  - Datová entita typicky prochází vývojem (myšleno asi tak že hodnoty atributů entity se mění)
    - Životní cyklus
  - Model = stavový diagram
    - Vazba na doménový / datový model
- CRUDL matice
  - Cíl: vědět, kdo manipuluje s jakými údaji
  - Create-Read-Update-DeleteList
  - Úroveň detailu
    - Analytická - uživatel, případ užití x informace
    - Návrhová - třída, funkce, proces x tabulka
- Úplnost modelu požadavků
  - Fáze zahájení projektu
    - Přesně cíl/vize projektu
    - Seznam klíčových aktérů, jejich cíle
    - Seznam/diagram podstatných případů užití (dle cíle)
    - Stručný popis klíčových PU, znalost klíčových vlastností
  - Fáze projektování
    - Kompletní seznam aktérů, popis důležitých
    - Kompletní specifikace, 80-100% funkčnosti, stručné povědomí o všech
    - Přesný popis mimofunkčních vlastností
- Shrnutí
    - Kontext - vize, aktéři
    - Funkčnost - případy užití, user stories, procesní model…
    - Struktury - doménový model, stavový
    - Vlastnosti - u PU, funkcí, doménových tříd, samostatně

Product backlog

- Obrysy požadavků agilně
- Společné rysy agilních specifikací požadavků:
  - Cílem je říct, produkt by měl umět tohle
- Forma
  - Epic, user story
- Obsah
  - Název, stručný popis, odhad pracnosti, testy
- **Vize produktu -> Feature (více než 2 sprinty) -> Epic (větší funkcionalita, > 1 sprint) -> Skupina user stories**

Modely základních struktur systému

Glosář

- Seznam důležitých pojmů
  - Klíčové
  - Nejasné
  - Sporné
- Stručný, všemi odsouhlasený popis = společný slovník, prevence nedorozumění
- Formát různý (word, excel, access…)

Doménový model

- Popis struktury problémové oblasti
  - Jaké jsou základní abstrakce používané v oblasti aplikace?
  - Jaké mají názvy, vzájemné vztahy a vlastnosti?
  - Jakým postupem je získáme?
  - Podle čeho si máme vymyslet stabilní třídy pro realizaci?
- Východisko = glosář
- Model = doménové objekty (diagram tříd)

Doménové objekty/třídy

- Doménové objekty - > třídy
  - Věci vyskytující se v problémové doméně
  - Klíčové pro fungování systému
  - Systém udržuje informace

- Podstatné aspekty
    - Terminologie uživatele, pojmy -> názvy tříd
    - Jen základní obrysy
    - Vztahy mezi třídami (asociace, kardinality)
    - Nezávislost na implementaci
- Jak je najít? Doménovou analýzou (konzultace, doménový expert, části systému podstatné z jejich pohledu - pomůže obrázek, rozhovor s uživatelem, pozorování práce)

Model nasazení

- Vztahy produktu k prostředí
    - Porozumění runtime a fyzickému prostředí
    - Charakteristiky, parametry
    - Odhad nákladů + podklad pro architekturu
- Alternativy
    - Zelená louka - součást návrhu architektury
- UML diagram nasazení

Strukturální model

- Postup se zaměřuje na data a jejich transformaci pomocí procesů systému,
- hlavními nástroji jsou tedy DFD popisující procesy a toky dat a ERD (Entity Relationship Diagram) popisující data a vztahy mezi nimi
- Zaměřuje se na vytvoření logického modelu nového navrhovaného systému (**esenciální model**) a následně přizpůsobení implementačním požadavkům (**implementační model**).
- Model **prostředí**
    - Definuje hranice systému a okolí
    - Obsahuje kontextový diagram a seznam událostí (event list)
- Model **chování**
    - Definuje vnitřní chování systému, tak aby plnil požadavky okolí
    - Používané modely (diagramy) DFD, ERD, DD (datový slovník), SP (specifikace procesů), případně STD (stavový diagram)

Strukturovaná analýza. Klasický postup "shora dolů". Chápe systém jako celek a postupně ho dekomponuje na jednodušší části. Je metodicky velmi dobře zvládnut. Nevýhodou je absence možnosti ukončit předčasně analýzu a implementovat konkrétní část systému. Typickým příkladem je metoda SSADM (Structured Systems Analysis and Design Method).

Objektový model

- Často se používá **modelovací jazyk UML**
    - Model případů užití
    - Doménový model
    - Diagram tříd
    - Stavové diagramy, sekvenční diagramy a další
- **CRC karty** (Class-Responsibility-Collaboration cards)
    - Umožňuje zvládnout návrh i složitých a velkých systémů
    - Není na první pohled vidět kudy vedou vztahy, musí se vyčíst z karet (barevné rozlišení, čáry na nástěnce…)
- Hierarchie objektů – sdružovány podle logických souvislostí do balíků a podbalíků
- Přirozený přechod od analýzy k návrhu

Doménová analýza

- Doménová analýza nepřihlíží podstatně k jednomu účelu a způsobu použití (kontextu použití), nýbrž shromažďuje pojmy a vazby pro doménu podstatné.
- Hledají se objekty, operace a vazby, které znalci z problémové oblasti pokládají za důležité (často používají jejich názvy apod.)

Objektově orientovaná analýza. Přístup "zdola nahoru", jehož základní myšlenkou je zobecňování. Začíná se z konkrétní části systému, která se analyzuje a popíše. V dalších částech se pak hledají společné rysy, z nichž se odvozují obecné zákonitosti. Výhodou je možnost rychlého paralelního vývoje prototypového software; nevýhodou je nedostatek ověřených metodických postupů a hlubší zkušenosti s rozsáhlejšími systémy.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Architektura softwarových systémů, význam a součásti architektury, formy popisu architektury, architektonické styly.

Wednesday, May 29, 2013 4:56 PM

Klíčová rozhodnutí

Proč - vize
Co - požadavky
Architektura
- Technologie
- Struktura
- Pravidla, konvence

Jak - návrh

Význam

- Klíčové aspekty organizace
  - Jak je systém členěn, jak a proč dělá to co dělá
  - Technologie
  - Hrubé členění nasubsystémy, závislosti a rozhrani mezi nimito spíš
- Aspekty prolínající celou implementací (konvence)
  - Jak systém navrhovat
  - Vzory návrhu, implementace
  - Mimofunkční aspekty…
  - RUP koncept "4+1" pohled
-

Co architektura dělá

- Architektura definuje konceptuální integritu systému.
- Systém má vždy právě jednu architekturu (může integrovat více stylů)
- Definice architektury je první krok návrhu
- Umožňuje myšlenkové pochopení návrhu velmi složitých systémů
- Stanovuje základní kameny návrhu a základní směry vývoje a údržby

Kontext, principy

- Kontext(daný)
  - okolí systému => vazby, důvody pro některá rozhodnutí
  - Stakeholders => úhly pohledu => aspekty architektury
    - Navíc oproti vizi a požadavkům
    - Aspekty: logická struktura, procesní pohled, varianty nasazení, datová integrace, bezpečnost, provoz a podpora, provozní infrastruktura, rozhraní
- Principy architektury
  - Efektivita - runtime, vývoj
  - Jednotnost
  - Srozumitelnost
- Výběr technologie, omezení
  - Programovací jazyk, databáze, knihovny
  - Použití frameworků a již hotových komponent
    - Make vs buy decision
  - Faktory ovlivňující/omezující výběr technologie
    - Smluvní podmínky
    - Standardy
    - Kontext
    - Marketing
    - Technické znalosti
- Validace architektury
  - Nutno ověřit užitečnost návrhu
  - Mechanismy
    - Návrh na základě klíčových use cases a mimofunkčních požadavků
    - Referenční architektura
    - Proof of concept implementace
    - Oponentura
- Dokumentace architektury
  - Referenční architektura
    - Dokument
    - Kostra aplikace
  - Dokument
    - RUP artifact: software architecture document
  - Modely
    - UML: implementation view (koomponenty), logical view (třídy, balíky), process view (interakce, stavový model)
    - Adhoc diagramy - visio, tabule
- Funkční členění

Architektura softwarových systémů, význam a součásti architektury, formy popisu architektury, architektonické styly.

Proč – vize
Co – požadavky
Architektura
- Použité technologie
- Struktura
- Pravidla, konvence

Jak – návrh

Architektura definuje funkční členění systému, závislosti jednotlivých částí systému a použité technologie. Architektura systému je vždy jen jedna.
Umožňuje myšlenkové pochopení návrhu velmi složitých systémů.

Kontext
- Okolí systému – vazby, důvody pro některá rozhodnutí

Výběr technologie
- Zkušenosti, existující komponenty, make vs buy decision, …

Validace architektury
- Nutnost ověřit užitečnost návrhu – proof of concept, oponentura, …

Dokumentace architektury
- Referenční architektura (MVC, MVVM)
- Dokument
- Model

Funkční členění architektury
- Subsystém, modul
- Modul je základní stavební jednotkou subsystémů. Subsystémy jsou funkčně soudržné celky často vázané na jednoho aktéra.

Formy popisu architektury
- UML (class diagram, component diagram,…), layer diagram

Architektonické styly
- Monolitická
- Dvouvrstvá (klient server)
- Třívrstvá – aplikační, prezentační a datová vrstva
- MVC – není třívrstvé, ale trojúhelníkové. Třívrstvá by byla MVVM.
- Vrstvená – více vrstev nad sebou
- SOA – service oriented architecture

- Subsystém, modul = skupina souvisejících prvků implementace tvořící funkční celek
  - Vícenásobně použitelné, volně vázané části
  - Celky vhodné pro vývoj a údržbu
- Subsystémy
  - Funkčně soudržné (lokalita změn)
  - Často vázané na jednoho aktéra
- Moduly
  - Základní stavební jednotka subsystémů
  - Snaha o vícenásobnou použitelnost
- Logická struktura subsystémů = moduly a jejich vnitřek, balíky tříd, třídy (low level architektura), jejich vztahy; Komunikace mezi moduly = ROZHRANÍ!!!!!!!!!!!!!

Součásti

- konvence a politiky (pravidla pro návrh, dodržují všichni vývojáři)
- **Funkční, procesní, datová, aplikační**
- členění, doménová analýza:
- **logické členění** (např. do balíků)
  - balík – skupina souvisejících tříd, tvořící organizační celek, mapování do jazyka (balík vytváří jmenný prostor), hierarchické vnořování
  - třídy v balíku funkčně příbuzné
  - vhodné protože bude přehled o systému a snadné rozdělení implementace mezi členy týmu
  - analytický model tříd je příliš rozsáhlý -> lepší jej členit
- **funkční členění do subsystémů**
  - subsystém = skupina souvisejících balíků a/nebo tříd tvořící funkční celek
  - vhodné, protože monolitická aplikace není praktická
  - jak najít subsystémy?
    - buď je to dopředu zřejmé (jednoduché, architektonické styly)
    - na základě objektového modelu (nutno vidět všechny třídy a vazby, pak shluk těsně vázaných tříd je kandidátem)
    - na základě případů užití

Formy popisu architektury

Modely - UML, Layer diagramy, …

UML, **Unified Modeling Language** je v softwarovém inženýrství grafický jazyk pro vizualizaci, specifikaci, navrhování a dokumentaci programových systémů. UML nabízí standardní způsob zápisu jak návrhů systému včetně konceptuálních prvků jako jsou business procesy a systémové funkce, tak konkrétních prvků jako jsou příkazy programovacího jazyka, databázová schémata a znovupoužitelné programové komponenty.
UML podporuje objektově orientovaný přístup k analýze, návrhu a popisu programových systémů. UML neobsahuje způsob, jak se má používat, ani neobsahuje metodiku(y), jak analyzovat, specifikovat či navrhovat programové systémy. Standard UML definuje standardizační skupina Object Management Group (OMG).

From <http://cs.wikipedia.org/wiki/Unified_Modeling_Language>

Architektonické styly

- **Vrstvení** - funkce jsou uspořádány do několika vrstev tak, že funkce vyšší vrstvy mohou využívat pouze funkcí podřízených vrstev.
  - **Monolitická**
  - **Dvouvrstvá (lehký/těžký klient)**
  - **Třívrstvá** je nejběžnějším případem vícevrstvé architektury.
    - **Prezentační vrstva**
    - **Aplikační vrstva (též Business Logic)**
    - **Datová vrstva**
  - **Porovnání třívrstvé s architekturou MVC**
    Model-view-controller má trojúhelníkovou topologii (ne třívrstvou) – pohled je obnovován (aktualizován) přímo modelem, na příkaz řadiče.

Architektura distribuovaných systémů
  - Klient-server
  - Peer to peer

Filosofický přístup k architektuře/ jednotící architektura: **Service-oriented architecture (SOA)**

Dle přednášek:

Základní architektonické styly

- **Klient server**
  - Tlustý klient
- **3-vrstvé a vícevrstvé**
  - Oddělení prezentace, business logiky a datové části
  - Dnes standard
- **Vrstvená**
  - Delegování na podřízené
  - Prezentace / řízení / doména / business služby / technická infrastruktura / knihovní třídy / systémové…
- **Další architektonické styly**
  - SOA - service oriented architecture viz SI
  - Pipes and filters
  - Blackboard
  - Broker

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Konfigurační management, jeho součásti a role ve vývoji software, základní postupy.

Wednesday, May 29, 2013 4:57 PM

Konfigurační management

„Proces identifikování a definování prvků systému, řízení změn těchto prvků během životního cyklu, zaznamenávání a oznamování stavu prvků a změn, a ověřování úplnosti a správnosti prvků.“ (IEEE)

= jak vytvářet, sestavovat a vydávat produkt, identifikovat jeho části a verze, a sledovat změny

- Administrační a manažerský aspekt Softwarového procesu

Prvek konfigurace

Configurable Item (CI)

- konstituující složka systému (konfigurace se sestává z prvků konfigurace)
- jsou atomické z hlediska změn a označování verzí, jednoznačně identifikovatelné
- př.: dokument, zdrojový soubor, knihovna, skript, spustitelný soubor, testovací data, …
- Je spravován SCM - ví se o jeho existenci, vlastníkovi, změnách, umístění v produktu…
- Je atomický z hlediska identifikace, změn
- Jednoznačně identifikovatelný - např. MSW/WS/IFE/SD/01.2
  - (typ prvku, označení projektu, název prvku, identifikátor verze…)

Konfigurace

SW konfigurace – souhrn prvků konfigurace reprezentující určitou podobu daného SW systému

- V konfiguraci musí být vše, co je potřebné k jednoznačnému opakovatelnému vytvoření příslušné verze produktu (včetně překladačů, build scriptů, inicializačních dat, dokumentace)
- Konzistentní konfigurace – konfigurace, jejíž prvky jsou navzájem bezrozporné (tj. zdrojové soubory lze přeložit, knihovny přilinkovat, …)

Role ve vývoji SW

Určení a správa konfigurace

- určení (identifikace) prvků systému, přiřazení zodpovědnosti za správu
- identifikace jednotlivých verzí prvků
- kontrolované uvolňování (release) produktu
- řízení změn produktu během jeho vývoje

Zjišťování stavu systému

- udržení informovanosti o změnách a stavu prvků
- zaznamenávání stavu prvků konfigurace a požadavků na změny
- poskytování informací o těchto stavech
- statistiky a analýzy (např. dopad změny, vývoj oprav chyb)

Správa sestavení (build) a koordinace prací

- určování postupů a nástrojů pro tvorbu spustitelné verze produktu
- ověřování úplnosti, konzistence a správnosti produktu
- koordinace spolupráce vývojářů při zpracování, zveřejňování a sestavení změn

Role ve správě změn

Change Control Board

- Skupina členů projektu, která má zodpovědnost za změnové řízení
  - Vyhodnocování a schvalování hlášení problémů
  - Rozhodování o požadavcích na změny
  - Sledování hlášení a požadavků při jejich zpracování
  - Koordinace s vedením projektu
  - Složení: jedinec - vývojář, QA osoba; tým - technické i manažerské role

Základní postupy

- **Identifikace konfigurace**: stanovení výchozího bodu (baseline, třeba každá major verze), známá kvalita (např. seznam bugů dané konfigurace, kompletní testování před stanovením výchozího bodu), kompletně opakovatelná
- **Řízení konfigurace**: rozhodování o způsobu změny konfigurace a koordinace změny konfigurace po schválení změny změnovým managementem (zm. mgmt schválí, chg. mgmt zavádí)
- **Sledování stavu konfigurace**: dokumentování stavu konfigurace každého vydání a změn konfigurace mezi vydáními (průběh změn mezi jednotlivými výchozími body)
- **Auditování konfigurace**: ověření, že nový výchozí bod implementuje všechny plánované a schválené změny, že nová verze je kompletní, a že dodávka je kompletní vč. právních opatření, dokumentace a dat.

! Role konfigurace ve vývoji vychází z postupů nebo naopak

ITIL: Konfigurační management IT infrastruktury podniku

- zaznamenávání informačních aktiv podniku, nastavení organizace a jejích služeb
- poskytování přesných informací o nastavení procesů a jejich dokumentace,
- poskytovat základ pro Incident Management, Problem Management, Change management a Release Management
- ověření konfiguračních záznamů oproti skutečnosti a jejich sladění.

Standardy

Standardů pro Configuration management existuje poměrně velké množství. Drtivá většina standardů je založena na metodice **ITIL**. Příklady některých standardů jsou: IEEE, ISO, ANSI, NATO standards.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Konfigurační management, jeho součásti a role ve vývoji software, základní postupy.

- **Konfigurační management (SCM)**
  - Proces identifikování a definování prvků systému
  - Řízení změn těchto prvků během životního cyklu
  - Zaznamenávání a oznamování stavu prvků a změn
  - Ověřování úplnosti a správnosti prvků
  - **= jak vytvářet, sestavovat a vydávat product, identifikovat jeho části a verze a sledovat změny**

- **Součásti SCM**
  - **Prvek konfigurace** = zdrojový kód, document, model, knihovna, spustitelný soubor, testovací data..
    - V SCM je prvek
      - atomický (dále nedělitený) z hlediska identifikace, změn
      - Jednoznačně identifikovatelný: typ, označení projektu, název, verze
  - **Softwarová konfigurace** = sestava prvků konfigurace rezprentující **určitou podobu** daného SW
    - *Konzistentní konfigurace* = konfigurace, jejíž prvky jsou navzájem bezrozpozné (lze přeložit)

- **Role SCM ve vývoji software**
  - **Určení a správa konfigurace**
    - Určení (identifikace) prvků systému, přiřazení zodpovědnosti za správu
    - Identifikace jednotlivých verzi prvků
    - Kontrolované uvolňování (release) produktu
    - Řízení změn produktu během jeho vývoje
  - **Zjišťování stavu systému**
    - Udržení informovanosti o změnách a stavu prvků
    - Zaznamenávání stavu prvků konfigurace a požadavků na změny
    - Poskytování informací o těchto stavech
    - Statistiky a analýzy (např. Dopad změny, vývoj oprav chyb, ...)
  - **Správa sestavení (build) a koordinace prací**
    - Určování postupů a nástrojů pro tvorbu spustitelné verze produktu
    - Ověřování úplnosti, konzistence a správnosti produktu
    - Koordinace spolupráce vývojářů při zpracování, zveřejňování a sestavování změn
- *Role ve správě změn: Change Control Board*
  - *Skupina členů projektu, která má zodpovědnost za změnové řízení*

- **Základní postupy**
  - **Identifikace konfigurace** – *stanovení výchozího bodu (baseline verze), známá kvalita (seznam bugů dané konfigurace => kompletně otestované), opakovatelná*
  - **Řízení konfigurace** – *rozhodování o způsobu změny konfigurace a koordinace změny konfigurace po schválení změny změnovým managementem => změnu mgmt schválí, chng. mgmt ji zavádí)*
  - **Sledování stavu konfigurace** – *dokumentování stavu konfigurace každého vydání a změn konfigurace mezi vydáními*
  - **Auditování konfigurace** – *ověření, že nový výchozí bod implementuje všechny plánované a svhálené změny, že nová verze je kompletní, a že dodávka je kompletní, dokumentace ...)*

Správa verzí, možnosti verzování, typické situace při správě verzí (větvení, značkování), nástroje pro správu verzí, vazba na správu změn.

Wednesday, May 29, 2013 4:57 PM

Správa verzí

správa verzí je součást úlohy konfiguračního managementu – identifikace konfigurace

účelem je udržení přehledu o podobách prvků konfigurace

- verze popisuje jednu konkrétní podobu
- v úložišti jsou skladovány všechny verze

druhy verzí

- evoluční = revize (př. Word 6.0)
- alternativní podoba = varianta (př. Word pro Macintosh)

určení konkrétní verze

- **verzování podle stavu** (verze prvku) – identifikují se pouze prvky
- **verzování podle změn** (identifikace změny prvku) – identifikují se také změny prvků, výsledná verze prvku vznikne aplikací změn

Možnosti verzování

granularita

- Jednotlivé prvky (verzování komponent)
  - Konfigurace nemá verzi
- Celé konfigurace (úplné verzování)
  - Verze konfigurace indikuje verze prvků
- Verze produktu

popis verze

- extenzionální verzování: každá verze má jednoznačné ID
  - major.minor + build schéma- např. 6.0.2800.1106 (MSIE 6)
  - kódové jméno: One Tree Hill (= Firefox 0.9)
  - marketingový: Windows 95
- intenzionální verzování: verze je popsána souborem atributů
  - např. OS=DOS and UmiPostscript = YES
  - C preprocesor umožňuje intenzionální stavové verzování - např. chceme variantu foo.c pro případ OS=DOS and UmiPostscript=YES

Možnosti verzování

- Rychlý přístup k jakékoli historické nebo alternativní verzi
- Možnost vytvoření branch, tagu => částečná izolace ale s možností aplikace vývoje v trunku
- Pojmenovávání milestones
- Celý tým má přístup k aktuálnímu stavu vývoje
- Aktuální stav vývoje je jednoznačně určen
- Soukromý pracovní prostor v rámci nejnovější nebo vybrané verze
- Možnost testování lokální změny a commitu až funkční a otestované součásti
- Možnosti pro řešení konfliktů
- Některé verzovací systémy jsou inherentně zálohovací (GIT)

Informace o verzi

- Identifikátor verze (extenzionální) - klíčovým požadavkem je jedinečnost
- "major.minor.micro + build" - např. 6.0.2800.1106 (MSIE6)
- Záleží na použitém nástroji a vztahuje se na prvky konfigurace
- Marketingové jméno se pak dává celému produktu (Longhorn)
- Další meta-data prvku jsou datum/čas vytvoření, autor, stav prvku/konfigurace, předchůdci…

Prostředí pro verzování: úložiště

- Úložiště (databáze projektu, repository) = sdílený datový prostor, kde jsou uloženy všechny prvky konfigurace projektu
  - Zdrojové kódy
  - Knihovny (přeložené) a kód třetích stran
  - Konfigurační soubory, datové soubory
  - Scripty pro build, testování a instalace
  - Dokumentace, modely, prototypy
  - Odpadkový koš
- Řízený přístup (udržení konzistence)

Práce s úložištěm

- Základní operace
  - Inicializace - vytvoření úložiště, naplnění bootstrap verzí projektu
  - Check-out - kopie prvku do lokálního pracovního prostoru
  - Check in (commit) - uložení změněných prvků do úložiště
  - Zjišťování stavu - sledování změn z úložišti versus v pracovním prostoru
- Přístup k zamykání při check in/check out
  - Read-only pro všechny
  - Pesimistický: read-write kopie prvku jen pro pověřeného
  - Optimistický: read-write pro kohokoli, řešené konfliktů

Pracovní prostor

- Workspace = soukromý datový prostor, v němž je možno provádět změny prvků konfigurace, aniž by byla ovlivněna jejich oficiální podoba používaná ostatními vývojáři

Typické situace při správě verzí

Správa verzí, možnosti verzování, typické situace při správě verzí (větvení, značkování), nástroje pro správu verzí, vazba na správu změn.

- **Správa verzí**
  - Součást úlohy konfiguračního managementu – identifikace konfigurace
  - Účelem je **udržení přehledu o podobách prvků konfigurace**
  - **Verze popisuje jednu konkrétní podobu**
  - V úložišti jsou **skladovány všechny verze**
  - Popis verze
    - **Extenzionální**: major.minor+build schéma, např. 6.0.280.1106 (MSIE 6)
    - **Intenzionální**: např. OS=DOS
  - Určení konkrétní verze
    - **Verzování podle stavu** (verze prvku) – identifikace pouze prvků
    - **Verzování podle změn** (identifikace změny prvku) – id. také změn prvků <- aplikace změn
  - *Granularita verzí: verze prvku -> verze konfigurace -> verze produktu*
- **Možnosti verzování**
  - Rychlý přístup k jakékoli historické nebo alternativní verzi
  - **Možnosti vytvoření branch** => částečná izolace, ale s možností aplikace vývoje v trunku
  - **Pojmenování milestonů = tag**
  - Celý tým má přístup k aktuálnímu stavu vývoje
  - Aktuální stav vývoje je jednoznačně určen
  - Soukromý pracovní prostor v rámci nejnovější nebo vybrané verze
  - Možnosti testování lokální změny a commitu až funkční a otestované součásti
  - **Možnosti pro řešení konfliktů**
  - Některé verzovací systémy jsou inherentně záhovací (GIT)
  - **Možnost paralelní práce na stejné konfiguraci**
    - Delta diff = množ.změn prvku konfigurace mezi dvěma po sobě následujícími verzemi
    - Větvení a spojování: trunk (hlavní), branch (paralelní), merge (sloučení hlavní + paral)
- **Typické situace při správě verzí (větvení, značkování)**
  - **Trunk**: hlavní vývojová větev (kmen)
  - **Branch**: větev – „soukromý“ vývojový prostor
  - **Merge**: sloučení větve do trunku a řešení konfliktů
  - **Tag**: značkování – označení milestounů, release, apod.
  - **Kolize a konflikty** – sloučení dvou konfliktních prvků
- **Nástroje pro správu verzí**
  - Centralizované:
    - **CVS** (Concurent Versions System) – práce s celými konfig., optimista při sluč. změn)
    - **SVN** (Subversion) – založen na CVS, verzuje změny (incrementální číslování revizí)
  - Decentralizované
    - **GIT** – pro nelineární vývoj, lokální repository
    - **Mercurial**, **Bazaar**
- **Vazba na správu změn**
  - Vazba revize na ticket/change request
  - Možnost požadavků na opravu / update konkrétních verzí

Trunk: Hlavní vývojová větev

Branch: Větve – „soukromý“ vývojový prostor

Merge: Sloučení větve a do kmene a řešení konfliktů

Tag: Značkování – označování milestounů, release apod.

Kolize a konflikty – diff, sloučení dvou konfliktních commitů…

Postup vývoje a verzování

- Check out výchozí verze
- Vývoj
- Lokální testování a opravy
- Check in nové verze
- Integrační testy a opravy
- Check in nové baseline

Codeline (vývojová linie)

- Je to série podob (verzí) množiny prvků konfigurace tak, jak se mění v čase
- Má přiřazena pravidla práce s codeline (kdy a jak je možno provádět změny…)
- Vrchol codeline obsahuje nejčerstvější verzi (head)

Tag (label)

- Označení konfigurace symbolickým jménem

Baseline

- Konzistentní konfigurace tvořící stabilní základ pro produkční verzi nebo další vývoj
    - Příklad: milník "stabilní architektura", beta verze aplikace
    - Stabilní: vytvořená, otestovaná a schválená managementem
    - Změny prvků baseline jen podle schváleného postupu
    - Při problémech návrat k baseline

Paralelní práce na stejné konfiguraci

- Důvod: velké úpravy, release, spekulativní vývoj, varianty…
- Cíl: vzájemná izolace paralelních prací tak, aby ukládané změny během nich neovlivnily ostatní (oddělení paralelních vývojových linií) -> cena za to je následné řešení konfliktů

Delta, diff

- Delta = množina změn prvku konfigurace mezi dvěma po sobě následujícími verzemi
- V některých systémech jednoznačně identifikovatelná
- Changeset: delta + důvod
- Diff a patch = rozdíl mezi verzemi (text, binární), aplikace rozdílu na verzi - viz changeset

Větvení a spojování

- Kmen (trunk, master) - hlavní vývojová linie
- Větev (brach) - paralelní vývojová linie - operace vytvoření větve = branch off, split
- Spojení (merge) - sloučení změn na větvi od kmene
    - Slučuje se delta od branch off nebo posledního merge
    - Řešení konfliktů: automatizace vhodná, ale ne vždy možná
    - 2-way a 3-way merge

Distribuovaný vývoj

- Geograficky distribuovaný tým, v čase distribuovaný tým, offline práce se synchronizací, experimentální lokální úložiště
- Možnosti: centrální úložiště s privátními větvemi, distribuovaný verzovací systém

Nástroje pro správu verzí

Centralizované

- RCS: revision control systém
    - Pesimista je to
    - Pracuje s jednotlivými soubory, nepodporuje projekty
    - Historie všech změn vč. autorů
    - Ukládá rozdíly
    - Umožňuje zamykání
- CVS: current versioning systém
    - Práce s celými konfiguracemi a projekty najednou
    - Optimista – slučování změn
- SVN: subversion
    - Velmi podobný CVS (následník)
    - Verzuje celé úložiště (inkrementální číslování revizí)
    - Souborová struktura

Decentralizované

- Každý uživatel má kompletní lokální kopii repozitáře (klony)
- Lokální commity, na centrální server lze nahrát víc commitů najednou
- GIT – jádro linuxu
    - Velmi nelineární vývoj, recenzování a začleňování
    - Nelze měnit historické verze
- Mercurial – Netbeans, OpenJDK, Symbian OS
- Bazaar – Ubuntu

- **Ruční verzování**
- **Základní** - správa verzí souborů
    - Obvykle extenzionální verzování modulů
    - Centrální úložiště
    - Ukládání všech verzí v zapouzdřené úsporné formě
    - Příklad nástrojů: rcs, cvs, subversion…
- **Distribuované**
    - Více úložišť, synchronizace
    - Flexibilnější postupy

    - Příklad nástrojů: SVK, git, Mercurial
  - **Pokročilé** - integrace do CASE
    - Obvykle kombinace extenzionálního a intenzionálního verzování
    - Automatická podpora pro check in/check out prvků z repository do nástrojů
    - Příklad nástrojů: ClearCase, Adele

Co nástroj má umět

- Operace s úložištěm
- Verzování
- Podpora týmu a procesu - vzdálený přístup, konfigurovatelné zamykání a přístupová práva, automatické oznamování, spouštění scriptů při operacích, Integrace do IDE, řádkové a webové rozhraní

Rcs

- Správa verzí pro jednotlivé textové soubory
- Ukládá historii všech změn v textu souboru
  - Informace o autorovi, datu a času změn
  - Textový popis změny zadaný uživatelem
  - Další info
- Používá diff(1) pro úsporu místa - poslední revize je uložena celá, předchozí jen pomocí diff
- Funkce: zamykání souborů, symbolická jména revizí, návrat k předchozím verzím, možnost větvení a merge

CVS

- Concurrent Versioning Systém
- Práce s celými konfiguracemi (projekty) najednou
- Sdílené úložiště + soukromé pracovní prostory
- Optimistický přístup ke kontrole paralelního přístupu (zkopíruj modifikuj sluč)
- Zjišťování stavu prvků, rozdílů oproti repository
- Integrace do mnoha IDE a CASE nástrojů

Subversion

- Následník CVS
- Bez omezení předchůdce - přejmenování, verzování adresářů, atomický commit, http přístup
- Nové možnosti - binární diff, meta-data, abstraktní síťová vrstva (DAV), čisté API
- Způsob práce a příkazy velmi podobné CVS
- Identifikace verzí - globální kontinuální celočíselné identifikátory - číslují commit

Vazba na správu změn

- Vazba revize na ticket/change request
- Možnost požadavků na opravu / update konkrétních verzí (např. long-term support)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Typy požadavků na změny, postup jejich zpracování, nástroje pro podporu řízení změn, vazba na správu verzí.

Wednesday, May 29, 2013 4:57 PM

Typy požadavků na změny

- Požadavek na novou funkci/vlastnost (Task)
- Chyba (Bug)
- Požadavek na změnu stávající funkcionality (Change Request)

Postup zpracování změny

- vytvoření/přijetí (přidělí se ID)
- vyhodnocení (možná řešení, jejich dopady a odhad pracnosti)
- rozhodnutí
  - způsob vyřízení (vyřešit/odmítnout/duplikát/odložit)
  - závažnost (kritická chyba/problém/vada na kráse/vylepšení)
  - priorita (vyřídit okamžitě/urgentní/vysoká/střední/nízká)
- přidělení odpovědné osobě / teamu
- zpracování
- uzavření
  - build: ověření konzistence; verzování: vytvoření nové baseline
  - Informovat zadavatele hlášení a další zájemce

Nástroje pro podporu řízení změn

- Bug tracking (BT) systémy
  - evidence, archivace požadavků (Ticket systém)
  - sledování stavu požadavku (BT, Ticket systém)
  - přehled, reporty, grafy, statistiky
  - realizace: emailové, webové, klientské
  - př. **Flyspray, Redmine, Mantis**
    - Jednoduché, snadná instalace, webové rozhraní, emailová notifikace
  - Př. **Bugzilla, Jira**
    - Robustní, pro velké projekty, konfigurovatelná
- Struktura projektu v BT systému
  - Název, kategorie hlášení, úrovně závažnosti, priority, verze produktu, operační systém, odhad a realita pracnosti…

Change Control Board (CCB)

- skupina členů projektu, která má zodpovědnost za změnové řízení
  - vyhodnocování a schvalování hlášení problémů
  - rozhodování o požadavcích na změny (může významně ovlivňovat podobu a chod projektu)
  - sledování hlášení a požadavků při jejich zpracování
  - koordinace s vedením projektu
- složení

- jedinec – vývojář, QA osoba
- tým – technické i manažerské role (vhodné, pokud má změna mít velký dopad)

Vazba na správu verzí

- Vazba ticketu/change requestu na verzi
- Vytvoření nové verze s opravou

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Sestavení produktu, postup sestavení a jeho varianty, nástroje pro sestavení.

Wednesday, May 29, 2013 4:58 PM

[aswi 04b]

Sestavení produktu = build

Řízení sestavení

Aktivity provádějící transformaci zdrojových prvků konfigurace na odvozené, zejména sestavení celého produktu

Cíl: vytvořit systematický a automatizovaný postup

Pojmy: build (integration, proces sestavení, sestavení)

Postup při vytváření sestavení

Build proces:

- míra formálnosti
- míra preciznosti

Kroky:

- příprava
- check-out
- preprocessing, překlad, linkování
- nasazení
- spuštění
- testování
- značkování, check-in
- informování

Vlastnosti sestavení

- **Jedinečnost a identifikovatelnost** – identifikátor jednoznačný, čitelný; vytvořitelný a zpracovatelný automaticky (schema pro id)
- **Úplnost** – tvoří kompletní systém, obsahuje všechny komponenty
- **Konzistence** – vzniklo ze správných verzí správných komponent – tj. konzistentní konfigurace
- **Opakovatelnost** – možnost opakovat build daného sestavení kdykoli v budoucnu (se stejným výsledkem)
- **Dodržujte pravidla vývojové linie** – build odpovídající baseline, zejména release, má striktní pravidla

Součásti prostředí pro sestavení

- Pravidla (neměnit) – vývojová linie, součásti a vlastnosti sestavení
- Scripty – check-out, značkování, check-in;preprocessing, překlad, linkování; nasazení, spouštění, testování; informování vývojářů, vytváření statistik; vytvoření distribuční podoby (packaging)
- Vyhrazený stroj a workspace – „build machine“

Varianty

Typy sestavení

- Co je použito pro sestavení (ušetřit čas na překladu):
  - Čistý
  - Úplný
  - Přírůstkový (inkrementální) build
- Účel sestavení (lokální/neoficiální komponenty povoleny)
  - Soukromý
  - Integrační (oficiální)
  - release build

Postup sestavení

Postupy

- **Základní postupy** – soukromé sestavení (private systém build) + sdílení součástí, integrační sestavení (integration build), release
- **Podpůrné aktivity** – smoke test, regression test, archvace prostřední, packaging
- Obecný cíl: odchytit co nejdříve okamžik kdy „se to rozbilo“

Soukromé sestavení

- Cíl: ověřit si konzistenci konfigurace – produkt lze sestavit pro mnou provedených změnách, předh check-in (problémy řeším já x všichni)
- Postup: sestavit produkt v soukromém prostoru
- Urychlení průběhu – použít inkrementální sestavení tam kde je to vhodné, vynechat postupy pro balení, vkládání info o verzi, pomoci si sdílením odvozených prvků (shared version cache)

Integrační sestavení

- Cíl: spolehlivě ověřit, že produkt jde sestavit – soukromý build nestačí (složité závislosti, specifika ve workspace, zjednodušení pro zrychlení)
- Postup: **celý produkt (vč. Závislostí) sestaven centrálně, automatizovaným a opakovatelným procesem**
  - postup co nejpodobnější sestavení pro release
  - maximální automatizace – typicky běží přes noc
  - mechanismy zaznamenání chyb a informování o nich
  - úspěšné sestavení může být označkováno ve verzovacím systému

Release build

- Význačné integrační sestavení: dodáno zákazníkovi (interní zákazník, např QA)
- Náležitosti release:
  - revize/verze konfigurace použité pro sestavení
  - datum vytvoření
  - identifikátor sestavení
  - další metadata: zodpovědná osoba, zdrojová značka konfigurace (z verzovacího systému), jakými prošlo testy (a výsledky), cesta k logům překladu (a testů)
  - „marketingová verze“ např Open cms 7.5

Diskuze o sestavení

- Celý proces automatizovat, plánovač spouštění buildu, vytváření čísel/identifikátorů sestavení, ukládání metadat do databáze a do verzování.
- Frekvence intragračního sestavení – čím častěji tím lépe – snažší nalezení chyb), kompromis trvání buildu x frekvence změn x velikost změn
- Samotné sestavení nestačí

Kusovník

- Kompletní seznam prvků sestavení
  - Reprodukovatelnost sestavení kdekoli, kdykoli

Sestavení produktu, postup sestavení a jeho varianty, nástroje pro sestavení.

- **Sestavení produktu**
  - **= Build**
  - Aktivity provádějící transformaci zdrojových prvků konfigurace na odvozené – zejména sestavení celého produktu
  - Cíl: vytv**ořit systematický a automatizovaný postup**
  - Pojmy: **build** (též itegration; proces sestavení; sestavení) – *proces a výsledek vytvoření částečné nebo úplné podoby aplikace*
  - Vlastnosti:
    - Jedinečnost a identifikovatelnost
    - Úplnost (tvoří kompletní systém, obsahuje všechny komponenty)
    - Konzistence (vzniklo ze správných verzí správných komponent = z konzistentní konfig.)
    - Opakovatelnost (možnost opakovat build daného sestavení se stejným výsledkem)
    - Dodržuje pravidla vývojové linie (odpovídá baseline, zejména release, má strikt. prav.)
  - Součásti: Pravidla (vývojová linie), Skripty (pro překlad, nasazení, testy, apod.), Vyhrazený stroj

- **Postup sestavení**
  - **= Build process**
  - Kroky:
    - (příprava)
    - Check-out
    - Preprocessing, překlad, linkování
    - Nasazení
    - Spuštění
    - Testování
    - Značkování, check-in
    - Informování

- **Varianty sestavení**
    - **Soukromé sestavení** + sdílení součástí = ověření konzistence konfigurace
    - **Integrační sestavení** = spolehlivě ověřit, že produkt jde sestavit
    - **Release** = dodáno zákazníkovi
  - Podpůrné aktivity
    - **Kusovník** a zapoudřená identifikace (seznam prvků sestavení)
    - **Zkouška těsnosti** (smoke test)
    - **Regresní testy**
    - **Archivace prostředí**
    - **Balení a distribuce** (packaging)
  - Nejlepší praktiky SCM + QA = (**continuous + daily build**) + (**smoke testy + regresní testy + unit testy**)

- **Nástroje pro sestavení**
  - **Skriptovací**: shell, perl, python, php, ...
  - **Buildovací**: make, ant, maven
  - **Verifikace sestavení**: xUnit (JUtnit, Cactus), testovací roboti

- Zejména při distribuovaném nebo jinak složitém buildu

- Samoidentifikující konfigurace pomůže
  - Znalost verzí bez přístupu k verzovacímu systému

Archivace prostředí

- Správa verzí objektů, které nejsou v úložišti
  - Nástroje, platformy, hardware, prostředí - identifikovat sestavení
- Klíčové pro dlouho žijící software (např. povinné v letectví)

Nejlepší praktiky: SCM + QA

- ověřené postupy sestavení pro největší zisk (zejména iterativní a přírůstkový vývoj)
  1. **Statické kontroly kódu**
     - Ověření formální správnosti + dodržování pravidel + metriky
     - Nástroje – překladač a jeho hlášení, C: lint, Java: pmd,findbugs
     - Postupy – programming by Contract, review, párové programování, automatický build- výběr
  2. **Jednotkové testy (unit testy)**
  3. **Pravidla pro codeline aktivního vývoje** (active development line)
  4. **Denní sestavení a zkouška těsnosti** (Daily build and smoke test)
     - Integrační sestavení + zkouška těsnosti – pravidelně 1xdenně (nočně)
     - Výsledky okamžitě reflektovány
     - Výhody: zvladatelné množství změn během denních check-in
     - Cena: trocha disciplíny, trocha automatizace
     - Smoke test = ověřit, že sestavení vytvořilo funkční produkt, vytvořit testy ověřující základní funkčnost, bez nároku na kompletní otestování
  5. **Regresní testy (regression test)**
     - Cíl: zajistit, aby nové funkce a vylepšení nesnižovaly již hotové kódu
     - Postup: ověřit build produktu pomocí testů, kterými již dříve prošel
     - Zdroj testů: chyby objevené QA, při validaci, zákazníkem
  6. **Soustavná integrace (Continuous integration)**
     - Dotažení do dokonalosti (nebo do extrému)
     - Klíčová je automatizace

Nástroje pro podporu sestavení

1. Scriptovací:
   - shell, perl, python, php
2. Buildovací:
   - make:
     - build (překlad a sestavení) projektu na základě popisu závislostí typu zdrojový - odvozený
     - makefile: definice pravidel (deklarace závislostí, příkazy pro překlad)
3. Maven (nebo Gradle):
   - deklarativní build
   - popis struktury projetku
   - build „automaticky“
   - Repozitáře, pluginy
4. Ant
   - Skriptování v xml
5. Hudson (Jenkins):
   - automatický build a průběžná integrace (vyhrazený stroj)
   - spuštění buildu
   - konfigurace buildu
   - informace
6. CruiseControl
7. Verifikace sestavení
   - xUnit(JUnit apod, Cactus)
   - testovací roboti
   - Indickej outsourcing

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Způsoby prevence chyb v software, metriky a oponentury.

Wednesday, May 29, 2013 4:58 PM

[http://www.robertdresler.cz/2012/02/techniky-pro-prevenci-softwarovych-chyb.htm]

[aswi přednáška 04b.pdf, aswi 06-kvalita.pdf]

Způsoby prevence chyb

- Cíl: zabránit vzniku a dalšímu šíření chyby
- Racionální proces a best practices
- Kontroly a měření meziproduktů (častější v úvodních fázích)+
    - **Automatizované testy**
        - Základní kontrola kvality kódu
            - Typicky unit testy
    - **Prověření meziproduktu nezávislým oponentem** (dříve než se z něj začne vycházet v další práci)
    - **Technická oponentura a podobné techniky**
        - Viz oponentury.
    - **Párové programování, refactoring**
        - Párové programování
            - Metafora řidič (udává směr, vysvětluje, naslouchá) + navigátor (dohledává, kontroluje, pomáhá) - svědomí páru (společný cíl, plné nasazení, komunikace)
        - Refactoring
            - Změna interní struktury software, která jej činí srozumitelnějším a snáze upravitelným, aniž by změnila jeho vnější chování
            - Detekce zapáchajícího kódu
            - Změna designu, oprava
    - **Strukturované procházení**
        - Podobné Faganovské inspekci, menší důraz na formálnost
    - **Peer review**
        - Kontrola nezaujatým čtenářem
        - Autor prochází kód a vysvětluje
        - Kolega hledá problémy a komentuje
- Měření
    - Kvantitativní ukazatele pomáhají najít slabiny kvality
    - Přesnost a dokazatelnost, možnost statistik
    - GQM přístup, FURPS

Detekční a opravné techniky

- Cíl: najít a opravit již existující chybu
- Testování a ladění (typické v koncových fázích, tzv. výstupní kontrola)

Psaní čistého kódu

- snižuje riziko "ukrytí" zákeřných programových chyb už v prvotní fázi psaní kódu
- podporuje efektivnější ladění

Refaktorizace kódu = změna struktury kódu, která nemá vliv na jeho celkovou funkčnost (přejmenování proměnné/metody, vyjmutí kódu do samostatné metody,..)

- snížení složitosti kódu
- Zpřehlednění

Revize kódu (Code Review)

Kdy provádět revizi kódu? Ideální by bylo během nebo těsně po vlastní implementaci. Revize kódu je jedním ze základních aspektů párového programování (Pair Programming). Dvojice společně vyvíjí kód a revize probíhá neustále. Pokud neaplikujete párového programování, můžete nastavit povinnou revizi kódu při umisťování změn na server. Revizi můžete také provádět v rámci celého vývojového týmu na pracovních poradách.

Jak je revize kódu efektivní? Záleží na úrovni zkušeností vývojáře a "revizora". U začínajících vývojářů budou revize častější a revizorem by měl být zkušený pracovník. Při revizi kódu vytvořeného zkušeným vývojářem nemusí být objeveno mnoho chyb, ale celý proces může posloužit k tomu, že posluchačům budou předány zkušenosti a praktické rady. Technik revizí kódu je více a liší se svojí formálností, obsazením revizního týmu a způsobem evidence nalezených defektů.

Statická analýza (kontrola) kódu = analýza kódu, která je prováděna bez nutnosti spouštění programu – soubor preventivních technik. Ověření formální správnosti + dodržování pravidel + metriky

- Nástroje: překladač a jeho hlášení
- Postupy: párové programování

Dynamická analýza kódu

Volba technologií

Automatizované testování

Testování je technika dynamické analýzy kódu. Psaní testů by podle metodologie Test-Driven Development (TDD) mělo předcházet vlastní implementaci. Zkuste si tento přístup zažít a uvidíte, že se vám zalíbí. A pokud ne, vězte, že testy stejně musíte napsat. Jinak se pro vás stane prvotní vývoj a především pozdější zásahy do produkčního kódu noční můrou. Testy jsou indikátory chybových stavů.

První typ testů, který napíšete, budou jednotkové testy (unit test). Základní logika jednotkového testu je jednoduchá. Voláte metodu instance testované třídy s určitými vstupy a validujete výstupy. Pokud výstupy neodpovídají předpokladům, test selže a je nutno hledat chybu v implementaci. Lze prohlásit, že vyšší pokrytí testy lépe pojistí váš kód. Ale je také nutno dodat, že testy se nesmí psát jen z formality, ale musí být správně cílené na problémové situace.

Pokud je test napsaný tak, že "vidí" do implementace, mluvíme o white-box testování. Pokud je testovaný kód pro test černou skříňkou, hovoříme o black-box testování. Oba přístupy mají své opodstatnění. Pomocí "bílého" testování snáze pokryjete všechny cesty provádění. "Černé" testování zase zosobňuje naivní přístup klientské strany, který může přinést nečekané způsoby volání.

Vyšší formou testů jsou integrační a systémové testy. Validují interakci více objektů a funkcionalitu větších celků. Pamatujte, že požadavky na kvalitu kódu testů jsou stejně přísná jako na vlastní testovaný (produkční) kód. Kód testů budete udržovat stejně dlouho jako produkční kód. Zjednodušeně shrnuto, testy by měly běžet krátkou dobu, na libovolném "kompatibilním" počítači, měly by po sobě uklidit a neměly by být na sobě vzájemně závislé.

Snažte se co nejvíce testů zautomatizovat, určitě se vám práce vyplatí. Bez automatizace nejsou některé typy testů vůbec proveditelné. Těžko se shání několik stovek uživatelů na zátěžové testy, kteří by v jednom čase začali používat a zatěžovat vaši aplikaci :)

Zautomatizovat se dá i interakce s uživatelským rozhraním vaší aplikace. Volba nástrojů závisí na technologii prezentační vrstvy.

Ruční testování

Automatizace buildů

Prototypování

Prototyp je funkčně zjednodušený základ (předobraz, demoverze) vyvíjeného systému. Měl by vzniknout relativně rychle a slouží k průběžné revizi požadavků. Prototyp můžete předvést

investorovi a získat zpětnou vazbu. Prototyp je vyvíjen v cyklech. V každém cyklu jsou zapracovány získané připomínky.

Revize výstupů v předimplementačních fázích

Motivace lidí

Metriky

[http://wiki.zvesela.cz/index.php/Zp%C5%AFsoby_prevence_chyb_v_software%2C_oponentury.]

Softwarové metriky:

- Cyclomatic Complexity = složitost části programu (např. metody) co do množství větvení a cyklů. Tyto programové konstrukce zvyšují počet možných cest provádění programu. Vysoké číslo indikuje komplikovaně napsané složité metody, které je problematické pokrývat testy. Řešením je metodu rozbít do více menších metod.
- Line Count (SLOC) vyjadřuje množství řádků kódu v metodách. Dlouhé metody jsou nepřehledné a dělají zřejmě více než jednu věc (pro danou úroveň abstrakce). Použitelnost takových metod je problematická, stejně jako pokrytí testy. Řešením je opět refaktorizace do více metod.
- Objektové metriky jako Weighted Method Count (celková složitost metod ve třídě), Depth of Inheritance Tree (počet předků třídy) a Coupling Between Objects (provázanost mezi objekty) mohou indikovat problémy v objektovém návrhu.

Z přednášek aswi 06-kvalita.pdf:

Formální verifikace

- Matematické důkazy správnosti
  - Návrhu
  - Implementace
- Model checking
  - Formální model systému - Petriho sítě, algebry (CSP) (a-priory nebo získaný analýzou kódu)
  - Model checker -> deadlock-free, liveliness, …
  - Soulad s implementací

Statistické kontroly

- Základ: metriky
  - Indikátor někde je něco špatně
- Stanovení očekávaných/správných hodnot
- Průběžné monitorování
- Korekce procesu, komponent při odchylkách

Z přednášek aswi 05-metriky.pdf:

Proč měřit

- Kvantitativní ukazatele
  - Pomáhají najít slabiny -> zlepšení kvality, přesnosti, efektivity…
  - Dávají přehled a kontrolu nad projektem-produktem - plán, kvalita, splnění požadavků…
  - Kalibrují odhady
- Výhody
  - Přesnost a dokazatelnost
  - Možnost statistik a vizuální prezentace

Metrika, měření

- Metrika = měřitelná charakteristika nějaké entity
- Je získána na základě dat (primitivních metrik)
- Měřený objekt - entita: produkt nebo proces
- Metriky samy o sobě "k ničemu" -> měření
- Plán měření - pro projekt
  - Co měřit, proč měřit, jak měřit
  - Jak s daty pracovat
- Organizational focus - záměr zlepšovat kvalitu

    - Vede k potřebě mít informace

Metriky produktu

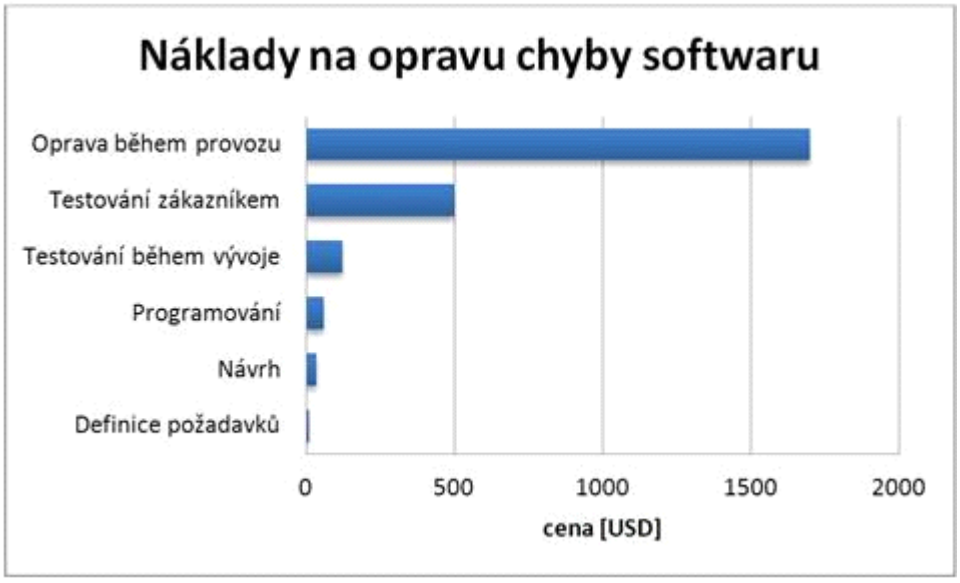
- Složitost, přehlednost
    - McCabe cyclomatic complexity
    - Fan-in / fan-out (afferent / efferent coupling) => stabilita
    - Weighted method per class
    - Lack of cohesion
- Velikost
    - Počet UC, funkčních bodů
    - Možná někdy případně i také LOC
        - SLOC, DSLOC, CBLOC, TLOC
- Metriky produktu (2)
    - Spolehlivost
        - MTBF = MTTF + MTTR
        - Dostupnost je pak (MTTF / MTBF) * 100
    - Kvalita (nepřímé metriky)
        - Pokrytí testy - kódu, požadavků
        - Charakteristiky defektů - hustota, výskyt
        - Kvalita zdrojového kódu
    - Nástroje
        - PMD, FindBugs, …
        - IDE pluginy
- **Projektové a procesní metriky**
    - Postup
        - Project velocity / burndown
        - Jitter - change requesty a jejich zpracování, staff turnover, změny postupu a plánu
    - Kvalita
        - Breakage = průměrná váha změnu LOC / CR (lines of code / change request)
        - Pracnost celkem, přepočtená na Change Requesty
        - Defect discovery rate, defect removal (zpracování, trendy)
- Jak měřit
    - **Top-down**
        - Definovat cíl měření
        - Zvolit metrik
    - **Goal-Question-Metric**
    - **Bottom-Up**
        - Jaké metriky?
- **Goal-Question-Metric**
    - Přístup k definování metrik
    - Rámec pro systém zaměřený na konkrétní problémy
    - **Goal** = problém + cíl měřícího programu
    - **Question** = měřené objekty a způsob měření
    - **Metric** = konkretizují získávaná data
- Nástroje pro měření
    - Spreadsheet, kalendář
    - Bugtracker
    - Statsvn
    - Junit a corbetura
    - Databáze
- Řízení měření
    - Plán měření
        - RUP template
        - GQM přístup
        - Definice metrik, jejich význam a zpracování
        - Způsob získání dat
    - Sledování projektu a produktu

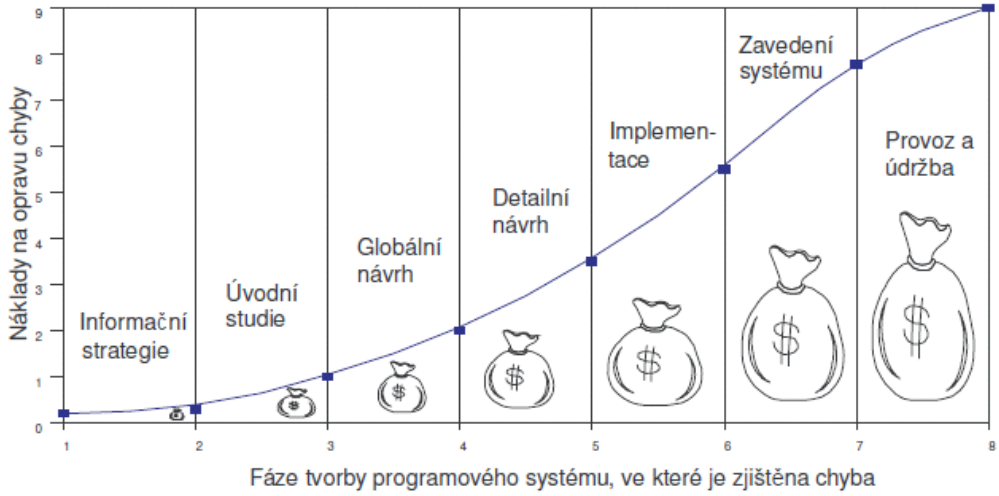
- Automatické získávání a vyhodnocování
- Sledování (management)
- Korektivní akce

Oponentura

Technická oponentura

- Technická oponentura
  - Též Faganovská inspekce
  - Skupinová technika, cílem je odhalit chyby v návrhu/kódu/dokumentu, sledování standardů, vzdělávání
  - Ne: dělat potíže autorovi (neúčast vedení), hledat nápravu chyb
- Role ve skupině
  - Moderátor - řídí diskuzi
  - Průvodce - předkládá dílo
  - Autor - vysvětluje nejasnosti
  - Zapisovatel - zaznamenává nalezené problémy
  - Oponenti - hledají chyby, obvykle podle seznamů otázek
- Postup
  - Příprava
    - Distribuce díla (moderátor), projití a hledání problémů (oponenti) - několik dní předem, cca 2 hodiny práce
  - Schůzka
    - Sekvenční procházení díla (průvodce či moderátor)
    - Vznášení připomínek
    - Zapisování nálezů (chyb a otevřených otázek)
      - Nejvýše 2 hodiny
      - Nepřipouštět dlouhé diskuze, řešení chyb (moderátor)
      - Možná následná schůzka pro vyřešení otázek
  - Závěry
    - Verdikt: v pořádku/drobné chyby/nutné přepracování/nová oponentura
    - Autor odstraní chyby dle nálezů, moderátor zkontroluje
      - **Dokument: Nálezy oponentury**
- Zhodnocení technické oponentury
  - Použitelné ve všech fázích životního cyklu
  - Velmi dobrá detekce chyb (až 75%)
  - Výsledkem jsou nižší náklady na vývoj a vyšší produktivita
  - Nároky
    - Náročné na čas
    - Je třeba zkušenost





Obrázek A 10 Náklady na změny v projektu

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Způsoby detekce chyb v software, metody testování, vztah k sestavení produktu.

Wednesday, May 29, 2013 4:59 PM

Způsoby detekce chyb

- Chyby ve zdrojáku – odhaleny při překladu
- Statické / Dynamické
- Ladění
- Testování
- Inspekce kódu
- Flow correctness (typ chyby - bacha na to)
- Formální verifikace – automatické ověření zda systém splňuje požadavek

Metody testování

- Whitebox
  - máme k dispozici zdrojové kódy programu, takže je to testování zaměřené na programovou logiku
  - **Unit testy** (testování malých částí programů, jako jsou podprogramy nebo třídy)
  - **Integrační testy** (jsou testovány komponenty a jejich interakce na základě rozhraní)

- Blackbox
  - Metoda testování bez znalosti kódu softwaru. Máme tedy k dispozici specifikaci softwaru a samotný software v podobě „černé skříňky", tzn. že se nemůžeme podívat dovnitř, jak funguje.
  - Řeší jiné typy chyb
    - nesprávné nebo zcela chybějící funkce
    - Chyby rozhraní
    - Chyby ve struktuře dat nebo externích databázích
    - Neočekávané chování
    - Chyby při inicializaci nebo ukončení
  - **Smoke test** (jestli to vůbec naběhne)
  - **Zátěžový test** (jestli se to sesype)
  - **Systémový test** (funkčnost v kontextu systému a interakce s jinými systémy)
  - **Hraniční testy** (vstupní data velmi blízko nebo na hranici akceptovatelnosti, v praxi je to totiž nejčastější zdroj problémů)
  - **Akceptační testy** (smoke test nového buildu a test zákazníkem po kompletním otestování)
  - **Usability testing** – testování uživatelem, hodnocení přívětivosti, intuitivnosti, …

  Beta testing sem nespadá – může se skládat z akceptačních testů, usability testů atd. poté, co se dosáhlo beta milestonu a produkt se v této fázi testuje

- Regresní testy - sada testů na základě specifikace. Pokud se vyskytne chyba, tato sada se obohatí pro budoucí testy

Vztah k sestavení produktu

Popsáno v jednotlivých metodách – různé typy v průběhu vývoje, před sestavením, po sestavení a před předáním, test zákazníkem. Pokud sestavením je myšlena verze produktu, tak při hlášení chyby je nutné verzi produktu uvést (to je ta vazba) aby bylo jasné, v jaké verzi problém hledat a zda už náhodou daný problém nebyl v nejnovější verzi vyřešen.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Měření software, produktové a procesní metriky, význam pro sledování kvality a řízení postupu.

Wednesday, May 29, 2013 4:59 PM

Metrika = způsob stanovení velikosti

Metriky software

- **Složitost, přehlednost**
  - počet možných cest skrz zdrojový kód
  - Fan-in / fan-out (afferent / efferent coupling) => stabilita
    - strukturální metrika, která měří poměr počtu modulů, které volají daný modul ku počtu modulů, které volá daný modul
  - Weighted Methods per Class
    - součet složitosti všech metod ve třídě
  - Lack of cohesion
    - nedostatek soudržnosti - jedna metoda dělái více funkcí (které se ve svém "smyslu" liší)
- **Velikost**
  - Počet Use Cases, funkčních bodů
  - Lines of Code
    - SLOC (Source Lines of Code),
    - DSLOC (Delivered Source Lines of Code),
- **Kvalita (nepřímé metriky)**
  - Pokrytí testy – kódu, požadavků
  - Charakteristika defektů – hustota, výskyt
  - Kvalita zdrojového kódu
- **Spolehlivost**
  - Střední doba mezi poruchami
  - dostupnost

Produktové a procesní metriky - viz předchozí otázka

Metriky produktu

- počet use case,
- počet podsystémů, modulů, tříd…,
- složitost modulů,
- počet řádků,
- datová velikost (ubuntu na 1 CD),
- počet odhalených chyb v jednotlivých modulech při testování,
- složitost dat modulu (funkční body),
- náklady na vývoj,
- člověkohodiny apod.

Metriky procesu

- Postup projektu
  - Rychlost vývoje
  - Change requesty a jejich zpracování (Burndown chart)
  - Staff turnover (fluktuace zaměstnanců),
  - změny postupu/plánu
  - …

- Kvalita
  - Breakage = průměrná váha změny (LOC (Lines of Code) / CR (Change Rate))
  - Pracnost celkem, přepočtená na CR (Change Rate)
    - Množství chyb (procenta) odhalených před odesláním zákazníkovi.

Metriky produktu

- Složitost, přehlednost
  - McCabe cyclomatic complexity
  - Fan-in / fan-out (afferent / efferent coupling) => stabilita
  - Weighted method per class
  - Lack of cohesion
- Velikost
  - Počet UC, funkčních bodů
  - Možná někdy případně i také LOC
    - SLOC, DSLOC, CBLOC, TLOC
- Metriky produktu (2)
  - Spolehlivost
    - MTBF = MTTF + MTTR
    - Dostupnost je pak (MTTF / MTBF) * 100
  - Kvalita (nepřímé metriky)
    - Pokrytí testy - kódu, požadavků
    - Charakteristiky defektů - hustota, výskyt
    - Kvalita zdrojového kódu
  - Nástroje
    - PMD, FindBugs, …
    - IDE pluginy
- Projektové a procesní metriky
  - Postup
    - Project velocity / burndown
    - Jitter - change requesty a jejich zpracování, staff turnover, změny postupu a plánu
  - Kvalita
    - Breakage = průměrná váha změnu LOC / CR (lines of code / change request)
    - Pracnost celkem, přepočtená na Change Requesty
    - Defect discovery rate, defect removal (zpracování, trendy)
- Jak měřit
  - **Top-down**
    - Definovat cíl měření
    - Zvolit metrik
  - **Goal-Question-Metric**
  - **Bottom-Up**
    - Jaké metriky?
- **Goal-Question-Metric**
  - Přístup k definování metrik
  - Rámec pro systém zaměřený na konkrétní problémy
  - **Goal** = problém + cíl měřícího programu
  - **Question** = měřené objekty a způsob měření
  - **Metric** = konkretizují získávaná data
- Nástroje pro měření
  - Spreadsheet, kalendář
  - Bugtracker
  - Statsvn
  - Junit a corbetura
  - Databáze
- Řízení měření
  - Plán měření
    - RUP template

        - GQM přístup
        - Definice metrik, jejich význam a zpracování
        - Způsob získání dat
    - Sledování projektu a produktu
        - Automatické získávání a vyhodnocování
        - Sledování (management)
        - Korektivní akce

Řízení postupu

- Plán měření
    - RUP template
- GQM (Goal Question Metric) přístup
    - Definice metrik, jejich význam a zpracování
- Sledování projektu a produktu
    - Automatické získávání a vyhodnocování
    - Sledování (management)
    - Korektivní akce

Význam pro sledování kvality a řízení postupu

- Lines of Code nic neznamená pro řízení kvality, ale třeba se dá odhadovat postup

- Je nutné sledovat kvalitu a upravovat vůči ní proces vzhledem k nákladům na zdroje, čas. Navíc pokud mineme chybu a vyjde do produkce, kromě řádově vyšších nákladů můžeme poškodit jméno společnosti

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Strategické řízení firem, poslání a role IT v organizaci, strategie IT/IS.

Wednesday, May 29, 2013 4:49 PM

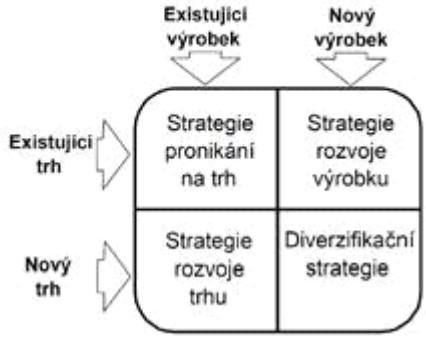
Strategické řízení firem

Strategii firmy je možné chápat jako **komplot**, neboli **plánovaný manévr**, nebo **model chování organizace** ve vztahu k jeho historickému vývoji nebo jako pozici, vyzdvihující význam výrobků dodávaných na specifický trh a konečně jako **charakter organizace**.

Strategie je **koncept**, **abstrakce** v myslích zainteresovaných stran.

Strategie je **perspektiva sdílení všemi členy organizace** (jedná se o kolektivní mysl, sjednocení jednotlivců ke společnému způsobu myšlení a jednání).

Typologie strategií (podle Ansoffa)



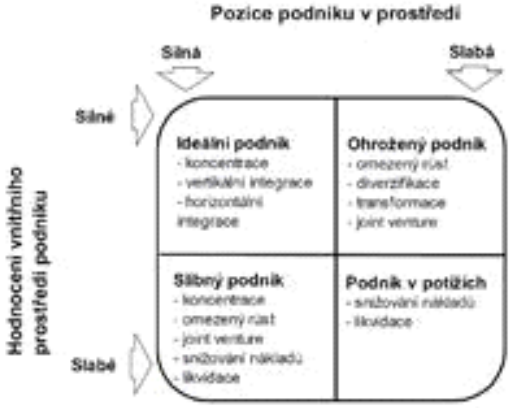
Strategické řízení je **vrcholovým řízením rozvoje podniku** jako celku v **delším časovém rozmezí**:

Strategické řízení = dlouhodobé plánování a směřování organizace

Proces určení dlouhodobých cílů a záměrů, přizpůsobení se podmínkám prostředí a alokace zdrojů organizace ve vztahu ke stanoveným cílům

Zaměření na rozsah činností podniku v dlouhodobém horizontu, které v ideálním případě vytvářejí soulad mezi podnikovými zdroji a měnícím se vnějším prostředím – zvláště trhem a zákazníkem

Integrovaný model strategických alternativ



Záměry

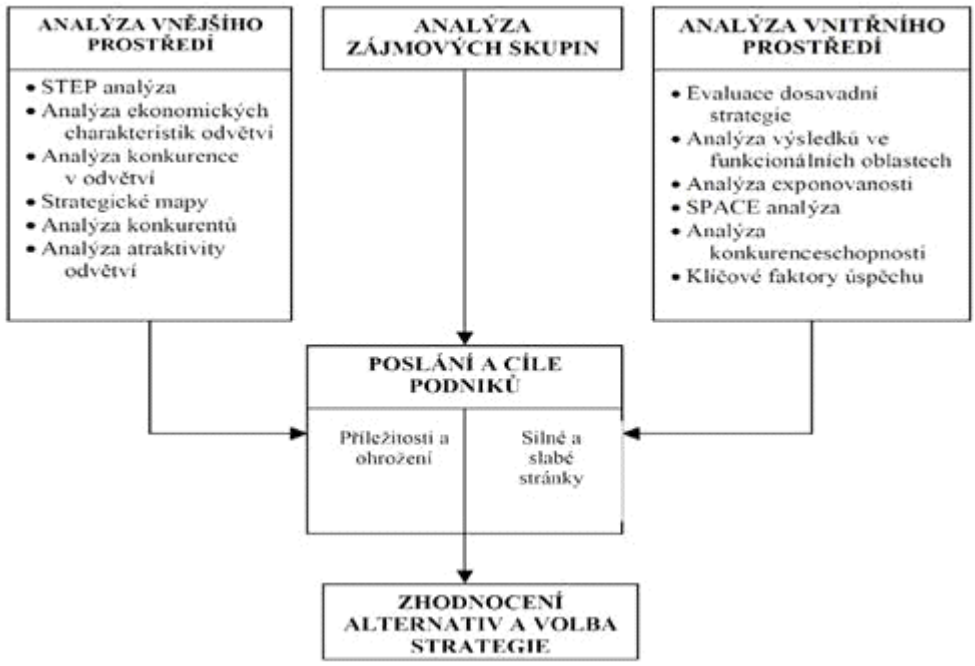
- Finanční i nefinanční zájmy různých zájmových skupin
- Umožňují a podporují zdůvodněné kompromisy
- Kompromisy u protichůdných cílů
- Motivující, ale dosažitelné
- Jdou napříč funkcionálními oblastmi

Cíle

- Operativní vymezení záměrů
- Vyjadřují, čeho chce podnik dosáhnout krátkodobě i dlouhodobě
- V souladu se zaměřením podniku

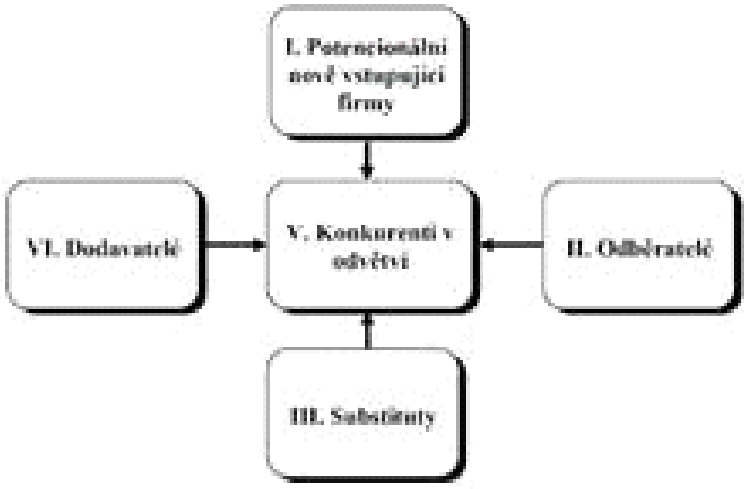
Analýza

Proces formulace podnikové strategie:



Analýza vnějšího prostředí

Porterova analýza:



STEP analýza

- Společenská (úroveň vzdělání, distribuce příjmů, životní styl…)
- Technologická (vládní výdaje za vědu a výzkum, nové vynálezy…)
- Ekonomická (trend vývoje HDP, inflace, nezaměstnanost…)
- Politická (Stabilita vlády, daňová politika, ochrana životního prostředí…)

Ptáme se přitom na otázky:

1. Které z vnějších faktorů mají vliv na podnik?
2. Jaké jsou možné účinky těchto faktorů?
3. Které z nich jsou v blízké budoucnosti nejdůležitější?

Analýza vnitřního prostředí podniku

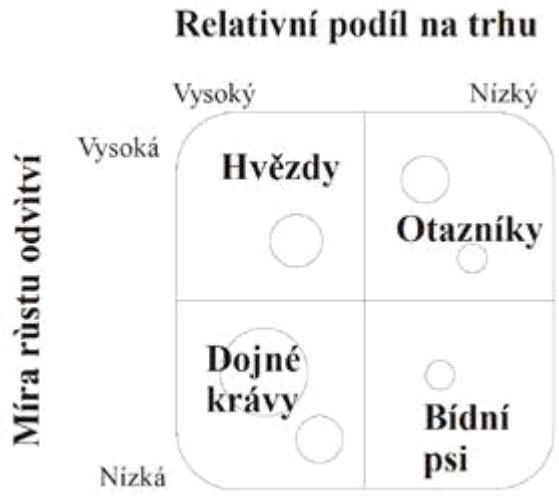
Analýza výsledků v jednotlivých funkcionálních oblastech: výroba, finance, marketing, úroveň řízení a lidské zdroje, výzkum a vývoj

Portfolio metody

Vytvoření matice portfolia, zmapování konkurenčního prostředí pro každou podnikatelskou činnost a vyvození závěrů o aktivitě všech položek portfolia, ohodnocení konkurenceschopnosti jednotlivých aktivit v portfoliu, hlubší proniknutí do situace podniku, určení potřeby finančních prostředků a dalších podnikových zdrojů na podporu strategií jednotlivých aktivit, porovnání aktivit z hlediska ziskovosti a přitažlivosti odvětví s následujícím roztříděním investičních priorit, kontrola s cílem vyhodnotit vyváženost portfolia, zjištění zda je portfolio v souladu s podnikovou strategií

Bostonská matice

Ukazuje spojitosti mezi tempem růstu obchodů a konkurenční pozicí společnosti, slouží především manažerům společností jako pomoc při řízení a dělání rozhodnutí ohledně zdrojů, dále ukazuje v oblasti skladového hospodářství v závislosti na financích, zajímavosti jako je prodej zboží na trhu, možnosti nárůstu či poklesu skladových zásob. 4 kvadranty:



Otazníky: výrobky zaváděné na trh vyžadují značné finanční vstupy, ale jsou šancí do budoucna, průzkum trhu rozhodne, jestli do nich dále investovat nebo je stáhnout.

Hvězdy: produkty s nejlepšími obchodními výsledky, udržení těchto výsledků je finančně náročné, ale výsledkem je vysoký zisk.

Dojné krávy: hlavní finanční opora firmy, přinášejí vysoké zisky bez větších finančních vkladů.

Bídní psi: produkty na konci prodeje, zvážení podniků, jak dlouho se vyplatí příslušný produkt udržovat na trhu a podporovat jejich prodej zesílenou marketingovou politikou.

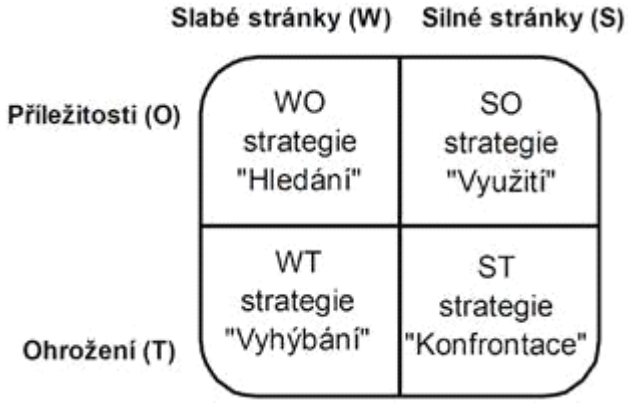
Analýza zájmových skupin

- Kulturní kontext – porozumění hodnotám, které společnost uznává
- Politický kontext – posuzujeme, jak různá očekávání jednotlivců nebo skupin mohou ovlivnit účel podniku. Ten se vyjadřuje v jeho poslání a cílech, na jejichž formulaci se podílí dominantní zájmová skupina.
- Etický kontext – týká se vlivu chování jednotlivců na hodnoty sdílené společností

Zájmové skupiny: akcionáři, věřitelé, zaměstnanci, zákazníci, dodavatelé, vlády, odbory, konkurenti, široká veřejnost

SWOT

SWOT analýza je základní metodou pro posouzení silných a slabých stránek podniku a příležitostí a ohrožení, která jsou závislá na vlivu vnějšího prostředí podniku. SWOT = Strengths, Weaknesses, Opportunities, Threats. Účelem analýzy je zaměřit se jen na ty stránky, které mají nějaký strategický význam.



Volba strategie

- Generování (vytváření) strategických alternativ
- Určení rámce problému
- Generování souboru

Metody pro podporu generování alternativ

- Generování scénářů
- Generování konfliktů
- Brainstorming
- Teorie chaosu
- Systémy podporující týmovou práci
- Zúžení souboru alternativ

Porovnání a hodnocení strategických alternativ

Hodnocení ve vztahu k následujícím krit.:

- Přijatelnost
- Vhodnost
- Realizovatelnost
- Poskytnutí výhody

Výběr alternativy jako budoucí strategie

Rozhodovací analýza, pro snížení chybovosti se využívá skupinové rozhodování

Kategorie alternativ

- **Zřejmé**, jasné alternativy
- **Kreativní** alternativy
- **Nemyslitelné** alternativy

Poslání

Poslání je integrální součástí strategického zaměření podniku, které vymezuje účel a smysl, kvůli kterému podnik existuje

V obecné rovině je to vize a mise podniku, v konkrétnějším vyjádření pak záměr a cíle.

Vize = vyjadřuje to, čím by podnik měl být – aspirace, zaměření do budoucnosti.

Mise = poslání = zformulovaná a napsaná vize + pohled do minulosti, proč firma vznikla

Efektivně formulované poslání – tržní orientace (vymezení podniku ve vztahu k trhu), realizovatelnost (optimální vymezení předmětu činnosti), motivace (zesilování pocitu zaměstnanců že jejich úsilí je významné a prospívá společnosti), specifikace (vyjádření hodnotového systému podniku, vztahu k zákazníkům, dodavatelům…)

Role IT v organizaci

core operational constituent that can improve business performance and increase shareholder value.

CIO je součástí boardu, ovlivňuje rozhodnutí společnosti.

http://www.iso.com/Research-and-Analyses/ISO-Review/The-Role-of-IT-in-the-Modern-Corporate-Enterprise.html

CTO = chief technology officer = kouká na to z pohledu technologie

CIO = kouká na to z pohledu procesu

IT (IS, Management, Procesní management…) má za cíl podporovat hlavní činnost podniku (projektu), přispívat k jeho úspěchu, umožňovat reagovat na hrozby a využívat (vytvářet) příležitosti. Řídí se strategií definovanou firemním / projektovým managementem, jeho cíle jsou vždy podřízeny cílům podniku/projektu.

Strategie IT/IS

Správně fungující IT prostředí, resp. informační systém, již v dnešní době není pouze konkurenční výhodou, ale je to klíčová podmínka pro business aktivity každé fungující společnosti. Důsledkem toho je krátkodobé i dlouhodobé plánování rozvoje IT prostředí, kterému zpravidla předchází stanovení strategie IT prostředí, ruku v ruce s business strategií společnosti.

Přistupovat strategicky k aktivitám, které mají závažnější dopad na rozvoj a provoz IT, nevyžaduje zpravidla časově náročné studie, bádání a nekonečné hledání. Je třeba mít na paměti, že lze využít řadu již vyzkoušených, ověřených a úspěšně dokončených scénářů. A to je i směr, který naše společnost preferuje – pochopit potřeby zákazníka a díky dlouholetým zkušenostem v různých IT prostředích, a znalosti vývoje trendů v IT, navrhovat směry rozvoje IT, které zákazníkovi budou přinášet hmatatelný užitek již nyní.

V hodnocení a návrzích strategie rozvoje IT se soustředíme na tyto priority:

- Zvyšování produktivity uživatelů
- Snižování nákladů na provoz IT (TCO)
- Zvyšování bezpečnosti zpracovávaných dat

http://www.mainstream.cz/cs/produkty-a-reseni/sluzby/strategie-it.aspx

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Komponenty podnikového IT, přehled oborových a technických standardů.

Wednesday, May 29, 2013 5:01 PM

Komponenty podnikového IT

- **Hardware**
  - Server, Workstation, Storage, Mobile, Monitoring, Backup, Asset Management (inventarizace)
- **Networking**
  - Connectivity, VPN, FireWall, Intranet, Extranet, Web, VoIP, Mobile device
- **Information**
  - Databáze, ETL (Extract Transform Load), MDM (Master Data Management), ECM (Enterprise Content Management), DMS (Document Managemtn System), DW (Data Warehouse), Business Inteligence, Business Analytics
- **Enterprise Applications**
  - ERP (Enterprise Resource Planing), CRM (Customer Relation Management), SCM (Supply Chain Management), HR (Human Resources), Helpdesk, Spisová služba, Accouting, Asset Management, CPM (Corporate Performance Management), Bussiness monitoring, Dashboardy
- **Security**
  - Identity management, Authentication, Autorization, CA (Certification Authority), Certifikáty, SSO (Single Sign-On), Antivir, Antispam
- **Colaboration**
  - E-mail, IM (Instant Messiging), Groupware, Calendar, Resource planning, DMS/CMS/ECM, Workflow, BPM (Business Process Management), Portal, Mash-up, Social network
- **Middleware**
- Application infrastructure (aplikační server, cluster, high availability, dissater recovery), ESB, messaging, IT governance, procesní servery, Business Rules, adaptéry, konektory, B2B gateways
- **Development**
- Requirements definition and management, Analysis, Design, Construction tools, Deployment, release management, testing, QA, Project management, Portfolio management

- Přehled standardů

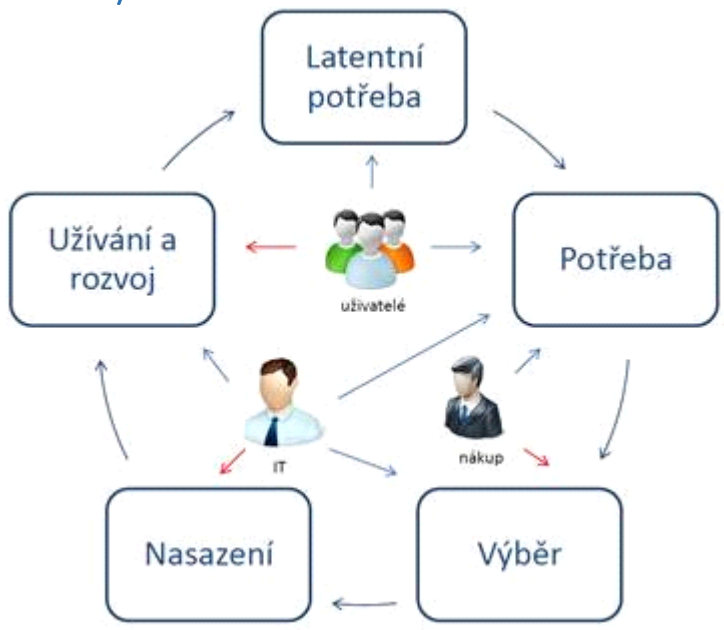
- **Technické**
- SQL, BPEL (Business Process Execution Language), BPMN(Business Process Model Notation), TCP/IP, DNS, SOA (Service Oriented Architecture)
- **Procesní**
- PMBOK (Project Management Body of Knowledge), ITIL (Information Technology Infrastructure Library), CMMI (Capability Maturity Model Integration), ISO
- TOGAF (The Open Group Architecture Framework) – komplexní přístup k návrhu, plánování, implementaci a dohledu enterprise architektury
- ISO 9000 (kvalita)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Životní cyklus IS, dodávka IS, proces akvizice IS/IT systému.

Wednesday, May 29, 2013 5:01 PM

Životní cyklus IS



Plánování – identifikace potřeb, vnitřní vlivy (strategie), vnější vlivy (legislativa), plánování financí, plánování zdrojů, koordinace projektů, způsob pořízení (krabicové řešení, na klíč, na míru…), Return of investment (ROI)

Analýza – detailní analýza potřeb, funkcí, parametrů; stakeholdeři, konzultační firmy, prováděna formou dokumentu, ustanovení realizačního týmu, pracovní skupiny, identifikace omezení (finance, lidé, čas)

Výběr řešení a dodavatele – poptávka, výběrové řízení (tender), Proof of Technology (**PoT**) = ukázka, demo; Proof of Concept (**PoC**) = pilot, placené, na míru společnosti; porovnávání řešení, vyhodnocování řešení, faktické vlastnosti, míra uspokojení požadavků, ekonomické faktory (cena, splátky, Return of Investment – **ROI** = celkový zisk/náklady, Total Cost of Ownership – **TOC** = cena včetně údržby apod.)

Dodávka IS

Implementace – může trvat dny až roky; dělí se na více fází, vyžaduje součinnost organizace;

možnost customizace, konfigurace, integrace na stávající systémy, nové a změněné komponenty – HW, SW, procesy, migrace dat, Quality Assurance (QA), testování, zaškolení lidí, předání

Postimplementační podpora – doladění systému, trvá obvykle týdny až měsíce

L1 podpora – uživatelská (helpdesk)

L2 podpora – systémová (admin)

L3 podpora – aplikační (změna kódu)

Vnější a vnitřní zdroje

Provoz, podpora – řádově roky, implementace dílčích změn, aktualizace, záplatování, sledování provozních parametrů, helpdesk, servicedesk, ITIL, reporting a sledování nákladů.

Ukončení, migrace – tzv. sunseting, dožití. Zmrazení investic a zahájení přípravy akvizice nového systému.

Proces akvizice IS/IT systému

Plánování – Analýza – Výběr řešení a dodavatele – dodávka a implementace – postimplementační podpora – provoz, podpora - dožití

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Proces výběrového řízení, poptávka a nabídka, výběr a nákup řešení, studie proveditelnosti, PoC, PoT, poptávkové řízení (RFI, RFP, RFQ).

Wednesday, May 29, 2013 5:01 PM

Proces výběrového řízení

Zadavatel nejdříve udělá RFI, aby zjistil, jaké jsou možnosti apod. Odpovědí na RFI je obvykle FS (Feasibility Study). Na základě toho zadavatel zpracuje poptávku (RFP, RFQ) a jako odpověď dostane nabídku. Z přijatých nabídek dle hodnoticích kritérií vybere zadavatel tu, která nejlépe splňuje požadavky uvedené v poptávce. Svou volbu pak oznámí ostatním účastníkům výběrového řízení.

Poptávka

- Žádost o kompletní nabídku
- Zadavatel chce realizovat nákup
- Má:
  - Formální část
  - Věcná část
  - Finanční část
- Desítky stran, někdy i stovky, přílohy…

Struktura

- Zadavatel
- Kontakty pro dotazy
- Harmonogram procesu
- Zadávací dokumentace
- Požadavky na uchazeče
  - Kvalifikační předpoklady
  - Reference
- Požadavky na nabídku
  - Formální – struktura jak má nabídka vypadat
  - Variantní řešení
  - Popis řešení
  - Strukturovaná cena

Nabídka

- Identifikace nabídky
- Disclaimer
- Kontakty na dodavatele (kdo napsal nabídku)
  - Obchodní
  - Technické
  - Statutární
- Informace o dodavateli
  - Přehled
  - Certifikace
  - Organizační struktura
  - Obraty
- Executive summary
- Popis variant řešení
  - Koncepce, architektura
  - Produkty
  - Služby
  - Customizace
  - Rozšiřitelnost
- Výhody řešení

- Diskuze požadavků zadavatele
- Zkušenost dodavatele, reference
- Navrhovaný harmonogram realizace
- Součinnost zákazníka
- Cenová nabídka
    - Co je a co není v ceně
    - Na pořízení
    - Na vlastnictví X let -> TCO (Total Cost of Ownership)
    - Platební kalendář
- Odkazy
    - Dokumentace použitých komponent třetích stran
    - Licenční ujednání
- Návrh smlouvy
- Složení projektového týmu včetně CV

Výběr a nákup řešení

Rámcový proces výběru a nákupu:

- Latentní potřeba
- Potřeba
- Studie proveditelnosti, analýza
- Vize řešení
- Požadavky
- RFI
- RFP – poptávka, zadávací dokumentace
- Výběr, shortlist
- RFQ – cenová nabídka
- Podpis smlouvy

Studie proveditelnosti

Zabývá se:

- Současný stav
- Seznam-analýza požadavků
- Možné přístupy řešení
- Popis variant
    - Funkcionalita, technologie, architektura, API, dokumentace
    - Zhodnocení variant
    - SWOT
- Analýza a mitigace rizik
    - Mitigace (co s nimi): ignorovat, přijmout, protiopatření, delegace
- Odhad nákladů a přínosů
- Provedené zkoušky, testy, prototypy
    - PoC, PoT

Obecná struktura

- Obsah
- Historie změn
- Úvod
- Účel
- Rozsah
- Reference
- Executive summary
- Obsahová část
- Shrnutí
- Přílohy

PoC

Proof of concept = pilotní verze řešení – placené, v součinnosti se zákazníkem

- Ověření vhodnosti řešení pro konkrétního zákazníka
- Ověření předpokladů TCO, ROI
- Identifikace problémových míst a rizik implementačního projektu
- Nastavení základu pro odhady pracnosti a složitosti
- Upřesnění požadavků zákazníka
- Data zákazníka
- Součinnost zákazníka
- Trvání dny až týdny
- Obvykle placené

PoT

Proof of technology = ukázka řešení, demo = zdarma většinou

- Technologické nebo produktové demo
- Ukázka funkčnosti (kompletního) řešení
- Ověření pro dané prostředí
  - Technické (OS, integrace…)
  - Tržní (segment, jazyk, velikost zákazníka…)
- Připravené dodavatelem
- Generická data
- Trvání hodiny – dny
- Obvykle zdarma

Poptávkové řízení

Rozdíl mezi poptávkovým z výběrovým řízením je, že v poptávkovém řízení oslovíme jen vybrané firmy, zatímco v tom výběrovém mají šanci všichni.

RFI

Request for information = business proces, jehož cílem je zjistit informace o možnostech dodavatelů, cílem je obvykle získání dostatečného množství informací, na jejichž základě je možné provést kvalifikované rozhodnutí.

Zadavatel hledá řešení, jedná se o první fázi RFP, odpovědí může být studie proveditelnosti, ceny jsou pouze orientační. Struktura RFI: zadavatel, harmonogram procesu RFI, RFP, definice problému, rámcové požadavky, omezení, cenová představa, kritéria výběru, časová představa – harmonogram realizace, požadavky na uchazeče – kvalifikační předpoklady a reference, struktura odpovědi. (z přednášek)

RFP

Request for proposal = proces, jehož cílem je vyžádání nabídky od dodavatelů. U RFP klient neví přesně co chce a nebo není schopen to, co chce dostatečně přesně specifikovat. Jinými slovy klient ví, jakou potřebu chce řešit, ale neví jak na to. Ještě jinak formulováno cílem je zjistit JAK by dodavatel řešil daný problém a KOLIK by si účtoval za implementaci svého řešení.

RFQ

Request for quotation = proces, jehož cílem je vyžádání nabídky od dodavatelů. Na rozdíl od RFP zde zadavatel přesně ví, co chce a také ví, že existuje více dodavatelů, od nichž si tu samou věc může koupit. Cílem je obvykle získání co nejnižší ceny. Ještě jinak formulováno cílem je zjistit pouze KOLIK by si dodavatel účtoval za konkrétní řešení.

http://www.negotiations.com/articles/procurement-terms

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Projektové a multiprojektové řízení, projektová kancelář, PMBOK.

Thursday, May 30, 2013 8:09 AM

Projektové a multiprojektové řízení

Co je podstatou projektového řízení? Tento výraz vznikl z anglického termínu project management, kterým se rozumí řízení projektu s jasně stanoveným cílem, který musí být dosažen ve stanoveném čase, nákladech a kvalitě. Projekty jsou často rozhodující součástí strategického řízení podniku. Mohou být zaměřeny na inovace výrobků, zavádění nových technologií, vývoj softwaru, modifikaci procesů a postupů, realizaci stavebních či investičních akcí, zavádění systémů řízení jakosti či realizaci podnikatelských záměrů. Jednoduše řečeno, projekt je organizované úsilí směřující k dosažení určitého cíle. Vyznačuje se jasně stanoveným konkrétním cílem, termínem, omezenými zdroji a především specifikací přínosů jeho realizace. Projektem naopak není periodicky se opakující práce či každodenní kontrolní činnosti.

Projektové řízení je komplexní proces, který vyžaduje profesionální přístup a podporu vrcholového managementu organizace. K řízení projektů se používají specifické nástroje a techniky. Všechny projekty se vyznačují společnými základními postupy a životním cyklem. Projekt je dynamický systém, který v sobě zahrnuje fázi iniciace a přípravy, plánování, realizaci i controlling projektu. Vyznačuje se samozřejmě i určitými specifickými riziky a v jeho průběhu se může vyskytnout řadu problémů. Jistá specifika mají i projekty v multiprojektovém prostředí, tedy v prostředí, ve kterém se řídí více projektů, dochází ke sdílení zdrojů a na projektové manažery jsou kladeny vyšší nároky. Neodmyslitelnou součástí profesionální projektového řízení a podmínkou úspěchu je i týmová spolupráce, protože právě lidský faktor se může stát příčinou mnoha komplikací nebo naopak zdrojem pozitivních synergických efektů.

http://www.systemonline.cz/clanky/aktualni-otazky-projektove-rizeni.htm

Projektová kancelář

Ve velkých společnostech se setkáváme s útvarem projektové kanceláře. Projektová kancelář hraje roli interního zákazníka ve společnosti.

Když stavíme projektový tým, musí v něm existovat duální hierarchie projektové organizace. Je zde strana zákazníka, jako příjemce dodávky a oponenta kvality dodávky, která se snaží minimalizovat finanční náklady a maximalizovat přínosy z ní plynoucí. Vůči ní vystupuje strana dodavatele, která se snaží rovněž minimalizovat své náklady a maximalizovat tím zisk, nebo tržby. Je to model klasického obchodního vztahu. Má dvě strany. Dodavatele, který se snaží rychle dodat, akceptovat a utéct a zákazníka, jehož role je kontrolovat, přebírat a platit.

Projektová struktura je virtuální. Vzniká a zaniká společně s projektem a je naplněna z interních liniových struktur zákazníka a dodavatele. Toto pravidlo se musí týkat všech projektů. Každý musí mít svého zákazníka a dodavatele, jinak by nemohl být úspěšný.

Pokud je ve společnosti rozhodnuto o tom, že bude implementován nový informační systém, nebo proběhne jiná velká dodávka od externího dodavatele, musí za sebe společnost postavit někoho, kdo odřídí projekt na straně zákazníka. V tomto případě není pochyb o tom, že je ideální, aby tuto roli sehrála projektová kancelář, neboť role projektového manažera vyžaduje potřebné kapacity a kompetence pro řízení projektu.

Dalším typickým případem, který může nastat, je realizace interního projektu. Stanovíme dodavatele, kterým je organizační jednotka, nebo tým složený z několika organizačních jednotek

společnosti současně. Zde to svádí k tomu, postavit do čela týmu dodavatele projektovou kancelář. Jenže kdo v tom případě sehraje roli zákazníka? Právě do této role by se měla projektová kancelář posunout. Projektového manažera dodavatele je nutné hledat mezi manažery se znalostí dodávaného produktu a zkušeností s vedením dodávek. Projektová kancelář pak má za úkol dohlížet na průběh projektu v roli zákazníka a odběratele, hodnotit jeho kvalitu a přejímat a kontrolovat výstupy.

A máme tu poslední případ. Společnost dodává svůj produkt externímu zákazníkovi. Jakou roli zde hraje projektová kancelář? Vážení přátelé, ideálně žádnou!

http://www.systemonline.cz/rizeni-projektu/role-projektove-kancelare-ve-spolecnosti.htm

A Project Management Office (PMO) is a group or department within a business, agency or enterprise that defines and maintains standards for project management within the organization. The PMO strives to standardize and introduce economies of repetition in the execution of projects. The PMO is the source of documentation, guidance and metrics on the practice of project management and execution. In some organisations this is known as the **Program Management Office** (sometimes abbreviated to **PgMO** to differentiate); the subtle difference is that program management relates to governing the management of several related projects.
Traditional PMOs base project management principles on industry-standard methodologies such as PRINCE2 or guidelines such in PMBOK.

From <http://en.wikipedia.org/wiki/Project_management_office>

PMBOK

Project Management Body Of Knowledge, je metodika a příručka pro projektové řízení vyvíjena neziskovou organizací zaměřující se na projektové řízení PMI (Project Management Institute). Základem je shromažďování nejlepších praxí z oboru a uvedení jich ve standard pro řízení projektů.

- Řízení integrace projektu
- Řízení rozsahu projektu
- Řízení času v projektu
- Řízení nákladů v projektu
- Řízení kvality projektu
- Řízení lidských zdrojů projektu
- Řízení komunikací v projektu
- Řízení rizik v projektu
- Řízení obstarávání v projektu

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Provoz IS/IT (dodávka a podpora IT služeb), řízení změn, ITIL.

Thursday, May 30, 2013 8:10 AM

Provoz IS/IT

POZN. Stejné jako 1.2.3???

Implementace – může trvat dny až roky; dělí se na více fází, vyžaduje součinnost organizace;

možnost customizace, konfigurace, integrace na stávající systémy, nové a změněné komponenty – HW, SW, procesy, migrace dat, Quality Assurance (QA), testování, zaškolení lidí, předání

Postimplementační podpora – doladění systému, trvá obvykle týdny až měsíce

L1 podpora – uživatelská (helpdesk)

L2 podpora – systémová (admin)

L3 podpora – aplikační (změna kódu)

Vnější a vnitřní zdroje

Provoz, podpora – řádově roky, implementace dílčích změn, aktualizace, záplatování, sledování provozních parametrů, helpdesk, servicedesk, ITIL, reporting a sledování nákladů.

Ukončení, migrace – tzv. sunsetting, dožití. Zmrazení investic a zahájení přípravy akvizice nového systému.

Řízení změn

POZN. ITIL – change management?

Vždy když se má zavézt změna do procesu/produktu/... musí se tyto změny zaznamenat – co se mění, jak, odpovědná osoba, očekávaný přínos... ITIL coby knihovna best-practices věnuje svou část v knize Service Support právě návrhu procesů Change managementu (tj. spadá to pod Operativní plánování)

- Zajistit hladkou a nákladově efektivní implementaci pouze schválených změn
- Minimalizovat vznik incidentů resultujících z provedených změn v infrastruktuře
- Change management odpovídá za:
  - Řízení oběhu Request for Change (žádosti o změnu),
  - Schvalování a plánování změn
  - Koordinaci implementace změn
- Change management svou činností zajišťuje FLEXIBILITU infrastruktury

ITIL

Co to je

Information Technology Infrastructure Library = soubor best practices pro IT service management = rozsáhlý, konzistentní a procesně orientovaný rámec pro IT service management. Je to knihovna řešící definici procesů, jejich I/O, stanovení rolí a odpovědností, měření kvality poskytovaných služeb, vazby mezi procesy, zásady pro implementaci procesů, přínosy procesu, náklady, critical success factors, zásady řízení a bezpečnosti ICT infrastruktury

Verze 2

Verze 3

Co řeší

Vydefinování procesů potřebných pro zajištění ITSM:

Stanovení cílů, vstupů a aktivit každého procesu

- Stanovení rolí a jejich odpovědností v daném procesu
- Způsob měření kvality poskytovaných IT služeb a účinnosti ITSM procesů
- Vzájemné vazby mezi jednotlivými procesy
- Postupy auditu a zásady reportingu pro každý proces
- Zásady pro implementaci procesů ITIL:

Přínosy každého procesu

- Critical success factors, možné problémy a vhodná protiopatření
- Náklady na implementaci a následný provoz
- Zásady pro řízení podpůrné ICT infrastruktury
- Zásady bezpečnosti ICT infrastruktury

Neřeší: konkrétní podobu organizační struktury, podobu a obsah pracovních procedur, projektovou metodiku implementace ITSM

Publikace = oblasti

Service Support a Service Delivery – základní, nejznámější, knihy o řízení, dodávce a podpoře IT služeb

Service Support

Service desk

Zajišťuje na denní bázi aktivní kontakt se zákazníky, uživateli, pracovníky vlastní organizace a pracovníky externí podpory, tzv. single point of contact pro uživatele a zákazníky
Zajišťuje obnovu standardní dodávky služby s minimálním dopadem na zákazníky, a to v mezích dohodnuté úrovně služby a podle obchodních priorit
Configuration management

Podporuje ostatní procesy poskytováním věrohodných informací o konfiguračních položkách infrastruktury
Stará se o konfigurační databázi CMDB
Incident management

Obnovuje normální provoz služby a to co nejrychleji při současné minimalizaci důsledků výpadku na provoz
Odpovědný za včasnou detekci problémů, jejich zaznamenávání a řízení jejich životního cyklu
Nezkoumá, proč k problémům dochází, jen hledá nejrychlejší řešení
Problém management

Zabránit opakování incidentů
Analyzuje incidenty, hledá příčiny, nápravu
Zajišťuje stabilitu celé infrastruktury
Change management

Zajišťuje hladkou a nákladově efektivní implementaci změn
Minimalizuje vznik incidentů plynoucích z provedených změn
Schvalování, plánování, koordinace a implementace změn
Release management

Zajistit hladký a kontrolovaný průběh nasazení nových verzí hardware a software do produkčního prostředí

Service delivery

Service level management

Udržování a zlepšování kvality IT služeb
Vyjednávání o obsahu a uzavírání Service Level Agreements, Operation Agreements…
Klíčový článek ITSM, spojuje poskytovatele a odběratele
Capacity management

Zajistit optimální kapacitu ICT infrastruktury
Hledání rovnováhy mezi existující kapacitou a náklady na upgrade
Availability management

Zajišťuje nákladově optimální dostupnost IT služeb, která bude v souladu s obchodními potřebami
Plánování, měření a sledování dostupnosti IT služeb
IT service continuity management

Obnova funkčnosti infrastruktury po vážném výpadku ve schválených mezích
Zpracování analýzy obchodních dopadů globálního výpadku
Financial management for IT services

Poskytuje nákladově efektivní správcovství ICT majetku a zdrojů

Sestavuje rozpočet ICT

ICT Infrastructure Management - Kniha aspektů řízení ICT infrastruktury od identifikace obchodních požadavků přes nabídkové řízení až po testování, instalaci, nasazení a následnou pravidelnou údržbu a podporu ICT komponent a IT služeb. Kniha popisuje hlavní procesy týkající se řízení všech oblastí souvisejících s technologiemi.

Application Management - Procesy celého životního cyklu aplikačního softwaru od prvotní studie proveditelnosti, přes vývoj, testování, vytváření aplikační dokumentace a školení uživatelů, implementaci do produkčního prostředí, provoz aplikace, změnová řízení během provozu aplikace až po stažení aplikace z používání.

Business Perspective - Určena zejména vedoucím pracovníkům obchodních a provozních úseků podniku. Jsou zde představeny základní prvky a principy řízení ICT infrastruktury, IT Service Managementu a Application Managementu, které jsou nezbytné pro podporu obchodních procesů.

Planning to Implement Service Management - Popisuje aktivity, úkoly a problémy související s plánováním, implementací a zlepšováním procesů IT Service Managementu v podnikovém prostředí. Je určena především členům implementačních týmů

Security Management - Popis organizace a řízení bezpečnosti ICT infrastruktury z pohledu IT manažera, a popis procesu plánování a řízení definované úrovně bezpečnosti informací a IT služeb včetně všech aspektů souvisejících s reakcí na bezpečnostní incidenty.

Software Asset Management - Popis procesů řízení, kontroly a ochrany softwarového majetku ve všech stadiích jeho životního cyklu

CCMDB – change and configuration management DB, info o všech konfigurovatelných Položkách

Integrace na datové vrstvě, MDM, ETL

Thursday, May 30, 2013 8:11 AM

Integrace na datové vrstvě

Přenos souborů

Jeden z nejjednodušších způsobů integrace na datové vrstvě je export, přenos a importu souborů. Rozhraní mezi systémy je v tomto případě realizováno samotným souborem, který má určitý definovaný formát, používaný jak zdrojovou, tak i cílovou aplikací. Výhodou je zejména skutečnost, že ukládání dat do souborů podporují všechny operační systémy a pro realizaci není nutné využívat další technologie.

Sdílená databáze

U tohoto stylu integrace používá více informačních systémů či aplikací jednu databázi pro společné ukládání provozních dat. Jednotlivé systémy tedy využívají společnou databázi a společný datový model. Výhodou oproti předchozímu stylu integrace pomocí přenosu souborů je aktuálnost dat, kdy nedochází k časovému zpoždění a veškerá data jsou vždy aktuální.

Sdílené soubory

V případě integrace pomocí sdílených souborů je architektura a princip podobný předchozímu stylu, tj. integraci pomocí sdílené databáze. Zde se však jedná o využití společného diskového úložiště a společných souborů obsahujících provozní data. Jedná se o jednoduchý způsob integrace na datové vrstvě, jehož problémem je zejména zajištění a ošetření souběžného přístupu k souborům.

Replikace dat

ETL

Viz níže.

http://www.ekf.vsb.cz/miranda2/export/sites-root/ekf/cerei/cs/okruhy/Papers/VOL13NUM03PAP05.pdf

MDM

Master Data Management, je přístup, který pomáhá jednoznačně identifikovat a integrovat klíčová data. Obsahuje procesy a nástroje pro definici a správu “master” dat (údaje z CRM,ERP, Data Warehouse).

MDM tvoří jeden z pilířů datové kvality. MDM je součástí Data Governance (data quality, data management, data policies). Cílem MDM (jako i celé DG) je zajistit kontrolované a konzistentní vytváření Master data a jejich kvalitu.

http://www.systemonline.cz/business-intelligence/master-data-management-1.htm

ETL

Extract Transform Load, mechanizmus získávání dat z provozních systémů podniku (ekonomika, skladové hospodářství, výroba, odbyt atd.), jejich následné zpracování a poskytnutí aplikacím pro podporu rozhodování (decission support systémy, datové sklady, business intelligence) - hlavně jde o uložení do cílového úložiště, LOAD do data warehouse typicky. Úkolem ETL nástrojů (IBM InfoSphere DataStage a QualityStage) je maximálně zjednodušit a zefektivnit implementaci a současně provoz ETL procesů.

Extraction – přenos dat ze zdroje (S-FTP, SCP, DB LINK – ODBC…)

Transformation – konverze dat, normalizace dat, unifikace (eliminace redundantních dat), historizace, agregace (při transformacích ve vyšších vrstvách skladů)

Loading – uložení do cílového úložiště

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Integrace na aplikační vrstvě, SOA.

Thursday, May 30, 2013 8:11 AM

Integrace na aplikační vrstvě

Služba

Služba je dobře definovaná a vymezená funkcionalita, která je zcela zapouzdřená a nezávislá na svém okolí (stavu ostatních služeb).

Web services

- Univerzální a platformně nezávislý způsob propojení na bázi XML.
- SOAP = **Simple Object Access Protocol** = nejběžnější způsob, jakým mezi sebou služby komunikují
- WSDL = **Web Services Description Language** – popis rozhraní služby
- Ws-gateways, ESB = metodiky propojování služeb

ESB

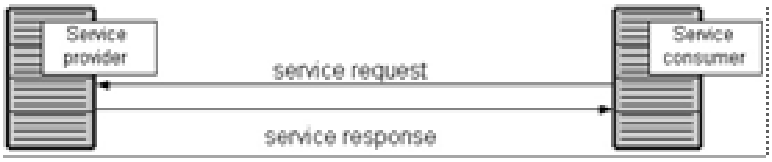
Enterprise Service Bus = koncept sběrnice služeb, místo propojování všeho se vším vytvoříme sběrnici, ke které vše připojíme. Komponenty (služby) totiž obvykle komunikují jeden s jedním a tvorba dvoubodových spojení pak vede časem k chaosu. ESB vytváří jakousi P2P síť. Jedná se o protokolově nezávislý způsob, jak vyvolat službu. Přijímá požadavky od WS klientů, zjistí, co s nimi má dělat, kam je předat, postará se o implementační detaily => ESB je implementací SOA.

Přínosy

- Úspora nákladů, dle statistik 30-40%, ale ne hned. Umožnění podnikům flexibilně reagovat na změnu, lépe zarovnat potřeby IT a business – k tomu nestačí jen web services, ale je potřeba mít i jiné prvky infrastruktury (middleware). ESB jako propojení služeb, Portál jako vhodné místo pro interakci uživatelů se službami.
- Služby jsou platformově i technologicky nezávislé
- Zjednodušuje využití ICT
- Umožňuje inkrementální nasazení
- Má schopnost rychle adoptovat změny (rychlost = úspora)
- Podporuje podnikání v reálném čase
- Jako vedlejší efekt přináší znovupoužitelnost služeb

SOA

Servisně orientovaná architektura (SOA) je soubor služeb, které jsou nějak spolu propojeny a vzájemně komunikují.



IS komunikují s jinými IS, spolupracují mezi sebou – aby mohly být používány rozumně, musí spolupracovat podobně jako služby reálného světa, tj. asynchronně reagovat na požadavky z různých zdrojů a být použity jako černé skříňky. SOA je navržena aby propojila mezi sebou libovolné služby.

Vývojový cyklus SOA: Posun od kódování ke skládání, Model -> Assemble -> Deploy -> Manage

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Integrace na prezentační vrstvě, portály, mashupy, web 2.0.

Thursday, May 30, 2013 8:11 AM

Integrace na prezentační vrstvě

Portály, mashupy.

Portály

Funkce

Enterprise portál webové místo, kde je pro určité skupiny uživatelů cíleně připraven nějaký obsah a funkcionalita (aplikace). = vylepšený webserver s novými funkcemi. Kombinuje různé aplikace a informační zdroje do jediné ucelené prezentace (AGREGACE). Uživatelé v různých rolích vidí odlišný obsah dle svých přístupových oprávnění (PERSONALIZACE). Uživatelé si mohou obsah sami přizpůsobit (CUSTOMIZACE).

Portlet

Stavební kameny stránek, jsou to vlastně kukátka do aplikací = rozšiřující zásuvné moduly.

Agregační princip

Kombinuje různé aplikace a informační zdroje do jediné ucelené prezentace (AGREGACE).

Mashupy

Aplikace, které kombinují výstupy z více různých služeb třetích stran (datových zdrojů) do jedné nové centrální aplikace, kde jsou tato data zobrazena. Touto na první pohled jednoduchou kombinací dat ze dvou zdrojů se získá snadno použitelný nástroj, který dává přidanou hodnotu datům z obou zdrojů. Jsou jedním z velkých fenoménů webu 2.0

Mashupy pro zaměstnance kombinují data z firemních znalostních databází jako jsou např. wiki spolu s dalšími vnitrofiremními aplikacemi a nebo s některou venkovní aplikací (mapy a jiné). Vznikají tak nové nástroje, které ulehčují orientaci v datech. Klientské mashupy mohou být postavené úplně stejně, ale budou kombinovat pouze data, která jsou veřejná a mohou být přístupná komukoli.

Web 2.0

Označení pro etapu vývoje webu, v níž byl pevný obsah webových stránek nahrazen prostorem pro sdílení a společnou tvorbu obsahu.

Termín "Web 2.0" označuje vývojovou fázi webu, kde se z počátků internetu, statického sdílení dat, vyvíjí dynamický web vytvářený samotnými uživateli.

Ke statickým HTML/CSS stránkám se přidávají dynamické programovací jazyky na serveru PHP, JSP, ASP a vznikají následující technologie:

- Wiki
- Sociální sítě
- Blogy
- Sdílení videa a fotografií

Při integraci Webu 2.0 do obchodních procesů podniků se hovoří o Enterprise 2.0.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Enterprise architektura, IT governance.

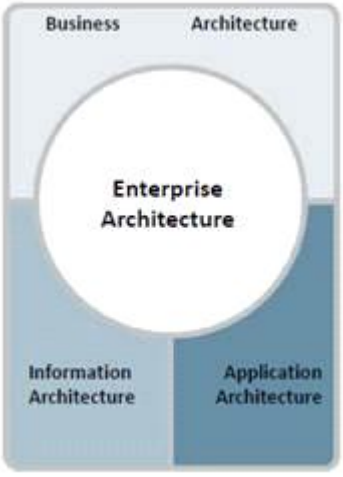
Thursday, May 30, 2013 8:12 AM

http://www.youtube.com/watch?v=qDI2oF1bASk

Enterprise architektura

Modelová situace - zaměstnanec potřebuje spojit data z více aplikací - přijde vývojář, napíše program/skript. Pak zaměstnanec zjistí, že by bylo dobrý tohle opakovat denně tak přijde admin a nastaví to aby se to spouštělo každý den. No a pokud se tohle zopakuje u hodně zaměstnanců, dostaneme něco čemu se říká "hairball architecture" chuchvalec vlasů. Problém nastane až se nějaký systém na který je tahle změť vazeb napojená zhroutí/přestane být podporován - pak je problém s tím cokoli udělat, protože nikdo neví kde co změnit. A tohle by měla řešit Enterprise Architektura. Ta by měla zjistit aktuální stav, a vytvořit plán, jak to vše přeorganizovat do nějakých smysluplných bloků, se kterými bude možné v budoucnu manipulovat - to je to "mapování IT na potřeby business". Nějaký plánovači to tedy naplánujou, pak přijde architecture review board a ti to musejí zkouknout že je to tak opravdu ok. No a pak se to implementuje. Cílem je umožnit změnu v podniku někdy v budoucnu bez zásadního dopadu na IT infrastrukturu.

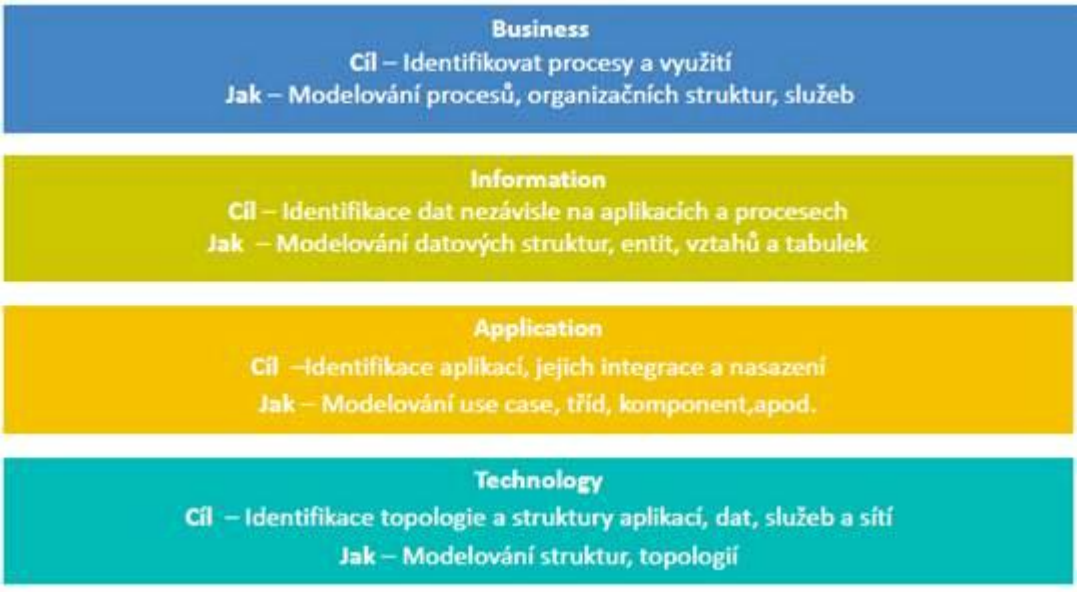
Enterprise architektura – modelování vztahu mezi organizací, byznysem a IT, koncepčně řízený rozvoj tohoto vztahu



EA pomáhá napasovat IT na potřeby business. Dále pomáhá aplikacím homogenních řešení, tzn. Opakovatelná řešení nějakých problémů – proč tisíckrát vymýšlet jak stavět strop když na to je postup který se dá použít vždy.

Součásti:

- Metodologie a metodiky
- Správa metadat
- Standardy
- Plánování IT, řízení projektů
- Modelování



Způsoby popsání Enterprise architektury:

- Zachmann Framework – taxonomie pro popis architektury systémů na enterprise úrovni

Zachman framework

	Why	How	What	Who	Where	When
Contextual	Goal List	Process List	Material List	Organizational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Organizational Unit & Role Rel. Model	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location details	Event Details

-

- TOGAF (The Open Group Architecture Framework) – komplexní přístup k návrhu, plánování, implementaci a dohledu enterprise architektury
- IBM EA Consulting Method – metodika IBM podporující kompletní řešení enterprise architektury a poskytující standardní výstupy popisující vlastní architekturu, dohled a koordinaci na programové i projektové úrovni, ohled a koordinaci realizace změn architektury

Složky

- Strategic capabilities network (SCN)
  - Identifikace kapacit zdrojů potřebných pro dosažení a naplnění strategických cílů
- Komponentní model (CBM)
  - Funkční model podniku
  - Podnik je popsán jako sada vzájemně propojených komponent
  - Komponenty jsou navrženy tak, aby byly schopné fungovat samostatně
  - „black box“ pohled – důležitá jsou rozhraní (poskytované služby, potřebné vstupy, vytvářené

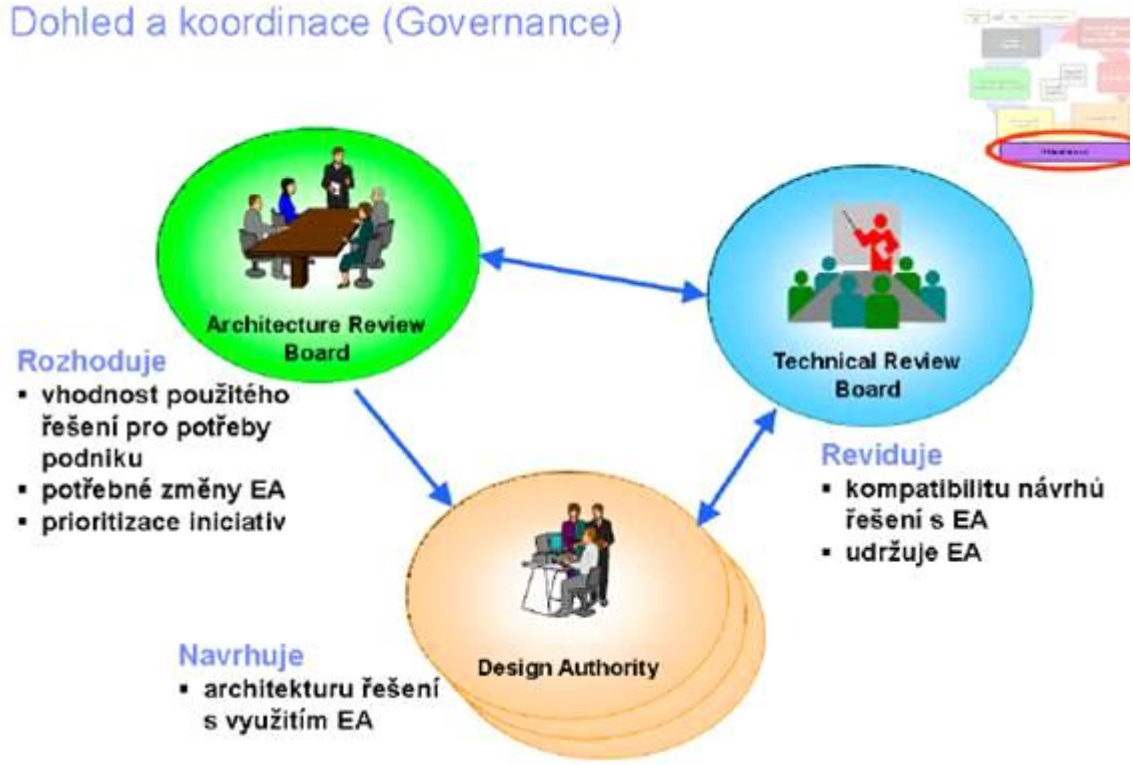
výstupy)

- Procesní model (BPM)
  - Procesní model popisující základní entity a vztahy mezi nimi
  - Události, aktivity, role (uživatelé) a data

IT Governance

Information technology governance is a subset discipline of corporate governance focused on information technology (IT) systems and their performance and risk management. The rising interest in IT governance is partly due to compliance initiatives, for instance Sarbanes-Oxley in the USA and Basel II in Europe, but more so because of the need for greater accountability for decision-making around the use of IT in the best interest of all stakeholders.
IT capability is directly related to the long term consequences of decisions made by top management. Traditionally, board-level executives deferred key IT decisions to the company's IT professionals. This cannot ensure the best interests of all stakeholders unless deliberate action involves all stakeholders. IT governance systematically involves everyone: board members, executive management, staff and customers. It establishes the framework (see below) used by the organization to establish transparent accountability of individual decisions, and ensures the traceability of decisions to assigned responsibilities.

From <http://en.wikipedia.org/wiki/Corporate_governance_of_information_technology>



Pěkná prezentace od Profinitu o EA (alespoň jsem z ní lépe pochopil ZACHMAN) str37
http://www.profinit.eu/fileadmin/Content/profinit.eu/Academy/invited-lectures/EnterpriseArchitectureInPractice.pdf

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Spolupráce a komunikace ve firmách – ECM, BPM, workflow, social business.

Thursday, May 30, 2013 8:13 AM

Spolupráce a komunikace ve firmách

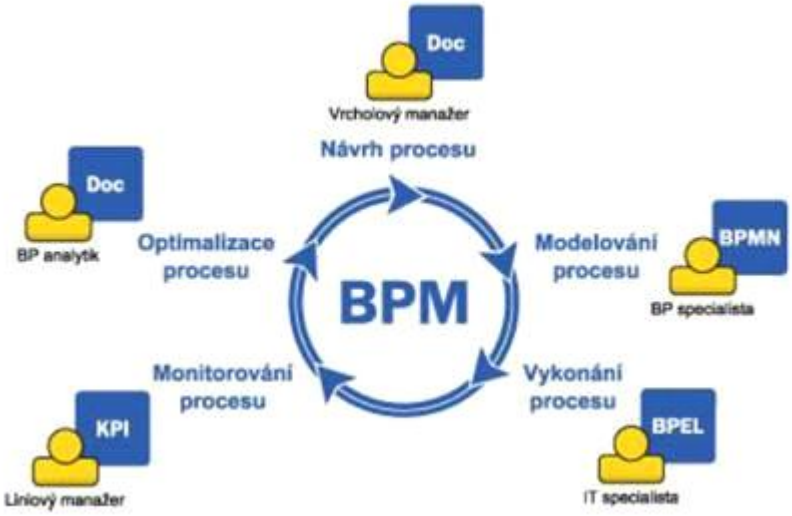
ECM

Enterprise content management, Řešení pro zpracování i nestrukturovaného obsahu (emaily,směrnice,podnikové znalosti). Smyslem je sdílení informací. Konverze dokumentů.

BPM

BPM – Business Process Management, správa podnikových procesů, systematický přístup ke zlepšování procesů v organizaci. Pomáhá zjednodušení a urychlení zavádění procesů v organizaci a jejich změn.

BP - Business Process. Je to abstraktní popis nějaké činnosti – běží dlouho, lidská interakce, platné stavy – nereálné provádět uvnitř aplikací, jež jsou odladěné a fungují jako černé skříňky, BP navrhují Business lidé a IT je implementují. Má jen jednoho vlastníka, který je za proces zodpovědný.

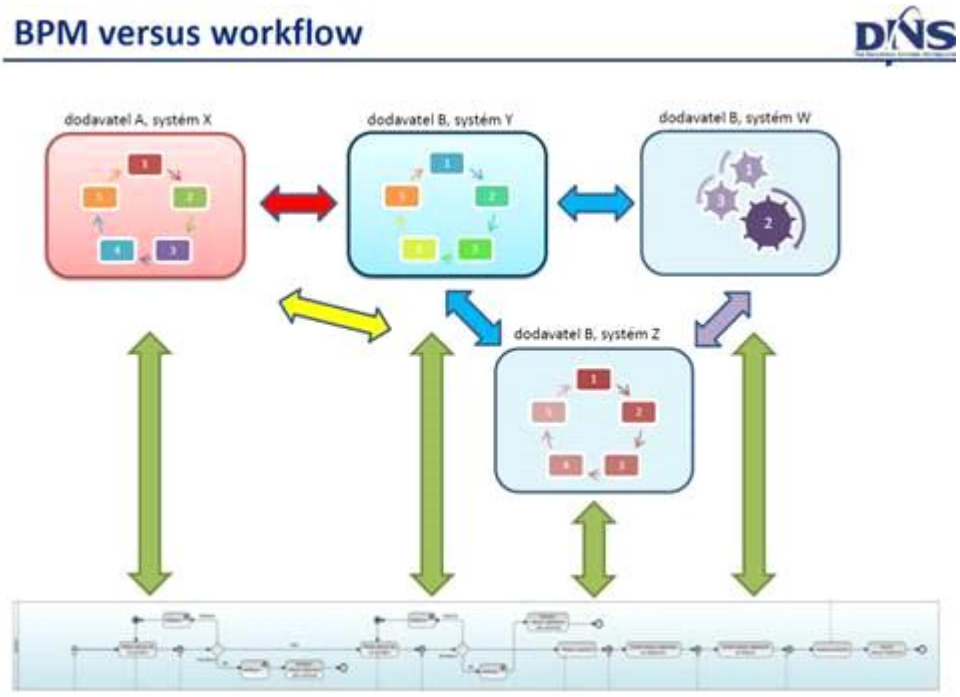


Principy BPM:

- Efektivní implementace a nasazení procesů ve firmě
- Přehledný diagram procesů
- Monitoring procesů
- Optimalizace procesů
- Zefektivňování procesů

Workflow

Workflow znamená automatizaci celého nebo části podnikového procesu, během kterého jsou dokumenty, informace nebo úkoly předávány od jednoho účastníka procesu k druhému podle sady procedurálních pravidel tak, aby se dosáhlo nebo přispělo k plnění celkových/globálních podnikových cílů.



Social business

Social Business využívá principy sociálních médií aby interně zlepšil organizaci (podnik) tak, aby byla více čiperná, průhledná a zapojená (nimble, transparent, engaged). Organizace, která implementuje moderní technologie (web 2.0) společně s organizačními, kulturními a procesními změnami, aby zvýšila svůj výkon a propojení s globálním ekonomickým prostředím.

Sociální podnik

- Hlubší vztahy se zákazníky, zaměstnanci, partnery, dodavateli
- Větší organizační transparentnost a agilitu
- Větší produktivitu a spokojenost zaměstnanců
- Větší zapojení a zpětnou vazbu od zákazníků
- Zrychlené inovace
- **KOMPETITIVNÍ VÝHODU**

Social business

- Naslouchání trhu, hledání advokátů (marketing, péče o zákazníky)
- Social je součástí procesů, propojení uvnitř i vně podniku (vývoj produktů a služeb)
- Vytváření komunit, rychlé drobné reakce (lidské zdroje, provoz, kancelář)

Tři roviny social business

- Spolupráce (nástroje social media)
  - Zapojení zaměstnanců, partnerů, zákazníků
  - Zrychlené generování nápadů
  - Rychlejší a lepší rozhodování
  - Lepší spolupráce
- Pochopení (analytické nástroje)
  - Směrování pozornosti, filtrování, polarizace
  - Pochopení vzorů chování, nálad
  - Metriky adopce a chování
- Transformace (nástroje procesní integrace)
  - Efektivita a zrychlování procesů
  - Rychlejší zapojení lidí
  - Podniková kultura inovace

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Outsourcing IT, ITaaS, cloud.

Thursday, May 30, 2013 8:13 AM

Outsourcing IT

Znamená, že firma vyčlení různé podpůrné a vedlejší činnosti a svěří je smluvně jiné společnosti čili sub-kontraktorovi, specializovanému na příslušnou činnost fáze outsourcingového procesu:

1) rozhodnutí o outsourcingu (outsourcovat ty činnosti, které nejsou pro podnik činnostmi strategickými),
2) detailní analýza části podniku určené pro outsourcing (slouží pro porovnání současných vlastních nákladů a dosavadní úrovně služeb s parametry nabízenými externí firmou – tzv. interní audit),
3) definice rozhraní podnik/poskytovatel (konkretizace požadované služby a určení návaznosti procesů na externě zajišťované činnosti)
4) výběr dodavatele.

ITaaS

IT as a Service – outsourcing IT, IT jako služba – je umožněno díky cloudu. Všeobjímající termín, který zahrnuje všechny modely uvedené níže.

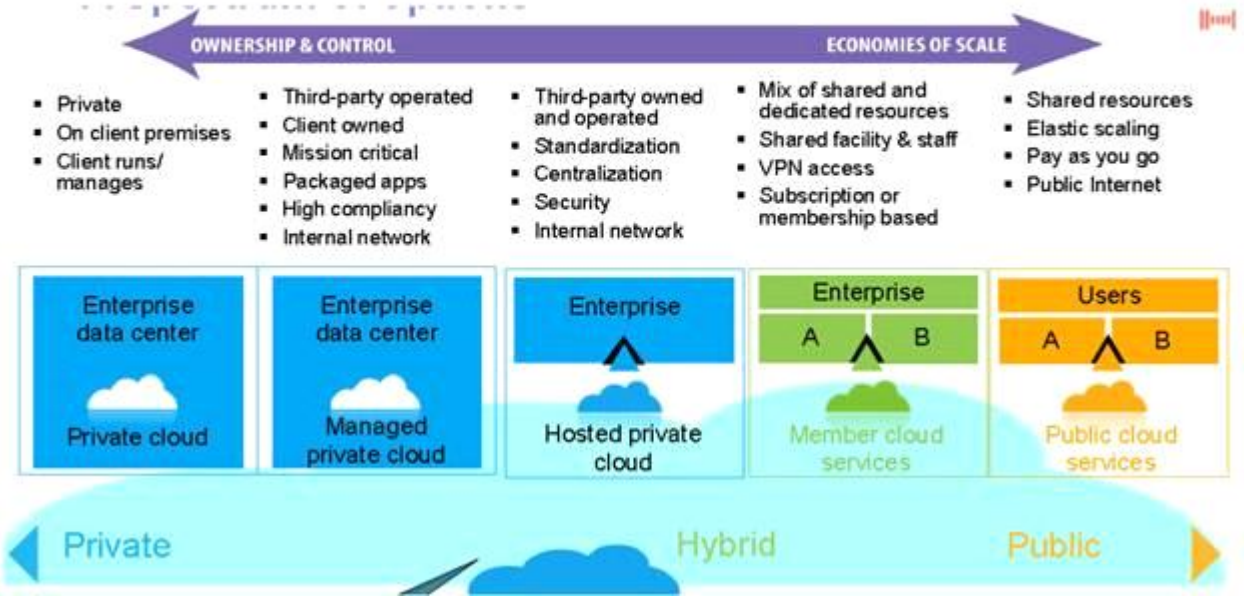
Cloud

Na Internetu založený model vývoje a používaní počítačových technologií

Distribuční model. Model se zabývá tím, co je v rámci služby nabízeno, obvykle software nebo hardware či jejich kombinace.

- **IAAS** — infrastruktura jako služba (z "Infrastructure as a Service") — v tomto případě se poskytovatel služeb zavazuje poskytnout infrastrukturu. Typicky se jedná o virtualizaci. Hlavní výhodou tohoto přístupu je to, že se o veškeré problémy s hardwarem stará poskytovatel. Na druhou stranu je někdy velice těžké toto akceptovat vzhledem k tomu, že hardware se bere jako něco, co vlastníme, na co můžeme sáhnout a jsme za to zodpovědní. IAAS je vhodné pro ty, kteří vlastní software (či jejich licence) a nechtějí se starat o hardware.
- **PAAS** — platforma jako služba (z "Platform as a Service") — poskytovatel v modelu PAAS poskytuje kompletní prostředky pro podporu celého životního cyklu tvorby a poskytování webových aplikací a služeb plně k dispozici na Internetu, bez možnosti stažení softwaru. To zahrnuje různé prostředky pro vývoj aplikace jako IDE nebo API, ale také např. pro údržbu. Nevýhodou tohoto přístupu je proprietární uzamčení, kdy může každý poskytovatel používat např. jiný programovací jazyk. Příkladem poskytovatelů PAAS jsou Google App Engine nebo Force.com (Salesforce.com).
- **SAAS** — software jako služba (ze "Software as a Service") — aplikace je licencována jako služba pronajímaná uživateli. Uživatelé si tedy kupují přístup k aplikaci, ne aplikaci samotnou. SaaS je ideální pro ty, kteří potřebují jen běžné aplikační software a požadují přístup odkudkoliv a kdykoliv. Příkladem může být známá sada aplikací Google Apps, nebo v logistice známý systém Cargopass.

Různé typy cloudů



Veřejný cloud (Public cloud): Cloud tak, jak je dnes nejčastěji chápán – tedy jako poskytování služeb IT (IaaS, PaaS, SaaS) prostřednictvím internetu třetí stranou, přičemž je zajištěna vysoká škálovatelnost a účtování podle využívaných zdrojů. Sdílen navzájem nesouvisejícími uživateli.

Soukromý cloud (Private cloud): Infrastruktura poskytující stejné služby jako veřejný cloud, ale pouze jedné organizaci. Je organizací většinou vlastněn a administrován. Aby bylo možno infrastrukturu označit za soukromý cloud, musí splňovat podmínku vysoké škálovatelnosti; někteří výrobci kladou důraz rovněž na schopnost účtování využívaných zdrojů jednotlivým složkám organizace.

Komunitní cloud (Community cloud): Cloud využívaný definovanou komunitou, například spolupracujícími firmami, komunitou vývojářů určitého projektu apod.

Hybridní cloud: Cloud složený z více různých cloudů, např. několika veřejných a soukromého (jejich **propojení**). Organizace může typicky využívat různé typy vzájemně propojených služeb od různých cloudových poskytovatelů.

From <http://www.businessit.cz/cz/cloud-computing-slovnik-pojmu-saas-paas-iaas.php>

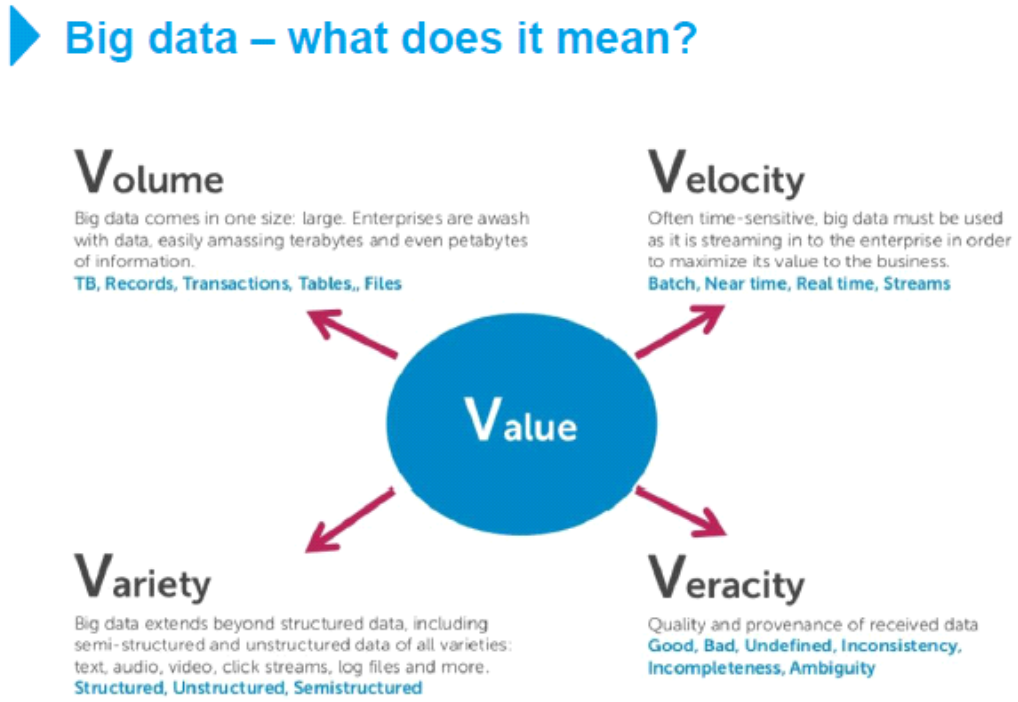
From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Aktuální témata podnikového IT – bigdata, social, mobile, analytics.

Thursday, May 30, 2013 8:14 AM

Bigdata

- Např. Twitter (12TB tweetů) nebo Monitoring telefonních hovorů v Číně
- Dat je moc, nelze udělat globální dotaz
- Nejde analyzovat všechna data
- Analýzu nelze provést v reálném nebo rozumném čase



Volume

- Dat je mnoho a jejich zpracování trvá příliš dlouho

Velocity

- Obrovské množství dat je potřeba analyzovat v krátkém čase

Variety

- Data mají různou strukturu a jsou nestrukturovaná: video, senzorická data, text…

Veracity

- Problematická důvěra v data

Využití Big Data

- Marketing
- Social network
- Analýza
- Skladování (dlouhodobé)

NoSQL databáze

- Obecný název pro řadu databázových aplikací zaměřených na vysoký výkon a škálovatelnost pro zpracování velikých objemů dat
- Jednoduché použití
- Horizontální škálování
- Hlavní kategorie:
  1. Key-value
  2. Columns
  3. Graph
  4. Document
- noSQL databáze pracují na odlišném principu než RDBMS – např. agregace dat do

dokumentů pomocí JSON (Java Script Object Notation)
- škálování standardních databází má své limity, noSQL databáze škálují jako skutečný distribuovaný systém
- wide column store – Hadoop
- document store – MongoDB, Crouchbase…
- key-value – Berkley DB

ACID

- Atomicity, Consistency, Isolation, Durability

Social

POZN. Viz Social Business???

Trend Web 2.0…

Social Business využívá principy sociálních médií aby interně zlepšil organizaci (podnik) tak, aby byla více čiperná, průhledná a zapojená (nimble, transparent, engaged). Organizace, která implementuje moderní technologie (web 2.0) společně s organizačními, kulturními a procesními změnami, aby zvýšila svůj výkon a propojení s globálním ekonomickým prostředím.

Sociální podnik

- Hlubší vztahy se zákazníky, zaměstnanci, partnery, dodavateli
- Větší organizační transparentnost a agilitu
- Větší produktivitu a spokojenost zaměstnanců
- Větší zapojení a zpětnou vazbu od zákazníků
- Zrychlené inovace
- **KOMPETITIVNÍ VÝHODU**

Social business

- Naslouchání trhu, hledání advokátů (marketing, péče o zákazníky)
- Social je součástí procesů, propojení uvnitř i vně podniku (vývoj produktů a služeb)
- Vytváření komunit, rychlé drobné reakce (lidské zdroje, provoz, kancelář)

Tři roviny social business

- Spolupráce (nástroje social media)
  - Zapojení zaměstnanců, partnerů, zákazníků
  - Zrychlené generování nápadů
  - Rychlejší a lepší rozhodování
  - Lepší spolupráce
- Pochopení (analytické nástroje)
  - Směrování pozornosti, filtrování, polarizace
  - Pochopení vzorů chování, nálad
  - Metriky adopce a chování
- Transformace (nástroje procesní integrace)
  - Efektivita a zrychlování procesů
  - Rychlejší zapojení lidí

Podniková kultura inovace

Mobile

BYOD – bring your own device – uživatel smí ve firmě využít vlastní zařízení
Důraz na SOA, zejména na RESTful služby (RESTful = webová služba splňující požadavky REST protokolu)
Vytváření mobilních aplikací pro firemní použití -> např. i analytics nástroje

Analytics

Potřeba analyzování sebraných statistik podle provozu procesů -> grafy, přehledy -> např. kokpit, nástroje BI (BI metriky, BA souhrn schopností, dovedností a znalostí), dashboardy (KPI = Key Performance Indicators)

BI

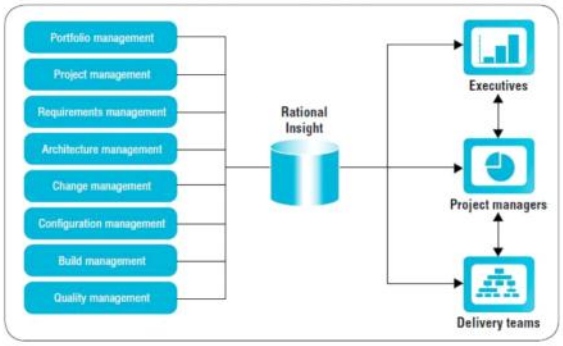
(Business Inteligence) - dovednosti, znalosti, technologie, aplikace, kvalita, rizika, bezpečnostní otázky a postupy používané v podnikání pro získání lepšího pochopení

chování na trhu a obchodních souvislostech

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/SI/Prednasky/SI-zkouska.docx>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Rational Insight – BI řešení pro vývoj SW



34

35

Datamining

Data mining ([dejta majnyn], angl. *dolování z dat* či *vytěžování dat*) je analytická metodologie získávání netriviálních skrytých a potenciálně užitečných informací z dat. Někdy se chápe jako analytická součást *dobývání znalostí z databází* (Knowledge Discovery in Databases, KDD),[1] jindy se tato dvě označení chápou jako souznačná.
Data mining se používá v komerční sféře (například v marketingu při rozhodování, které klienty oslovit dopisem s nabídkou produktu), ve vědeckém výzkumu (například při analýze genetické informace) i v jiných oblastech (například při monitorování aktivit na internetu s cílem odhalit činnost potenciálních škůdců a teroristů).

From <http://cs.wikipedia.org/wiki/Data_mining>

„Vnitřní“ programovací konstrukce (Embedded SQL) - procedurální prostředky v rámci jazyka SQL

Wednesday, May 29, 2013 4:49 PM

Embedded SQL

Přímý přístup programovacího jazyka do databázových struktur – **jazyk (překladač) je obohacený o konstrukce pro SQL**

Je metoda vkládání řádkových SQL příkazů do kódu programovacího jazyka, protože tento programovací jazyk neumí parsovat SQL. Vkládané SQL je parsováno v SQL preprocesoru.

- **Query Language**
  - http://people.ku.edu/~nkinners/LangList/Extras/classif.htm
- SQL dotazy jsou psány přímo ve zdrojovém kódu. Syntaxe SQL je přizpůsobena syntaxi daného programovacího jazyka.
- Před kompilací programu se musí zdrojový kód zpracovat preprocesorem Embedded SQL, kdy SQL dotazy jsou nahrazeny odpovídajícím kódem programovacího jazyka.
- Nejčastěji v kombinaci s jazyky C/C++.

Podpora v databázích

Podporují klasicky velké databázové systémy

- IBM DB2 (C/C++, Java, Cobol)
- Oracle (Cobol, *Pro*C* - embedded SQL Oraclu pro C/C++)
- PostgreSQL (C/C++)

Nepodporují

- MySQL
- Microsoft SQL Server (starší verze podporovali C)

Příklad syntaxe

Oracle Embedded SQL v jazyce C:

```
    {
        int a;
        /* ... */
        EXEC SQL SELECT salary INTO :a
                 FROM Employee
                 WHERE SSN=876543210;
        /* ... */
        printf("The salary is %d\n", a);
        /* ... */
    }
```

Embedded SQL

The first technique for sending SQL statements to the DBMS is embedded SQL. Because SQL does not use variables and control-of-flow statements, it is often used as a database sublanguage that can be added to a program written in a conventional programming language, such as C or COBOL. This is a central idea of embedded SQL: placing SQL statements in a program written in a host programming language. Briefly, the following techniques are used to embed SQL statements in a host language:

- Embedded SQL statements are processed by a special SQL precompiler. All SQL statements begin with an introducer and end with a terminator, both of which flag the SQL statement for the precompiler. The introducer and terminator vary with the host language. For example, the

introducer is "EXEC SQL" in C and "&SQL(" in MUMPS, and the terminator is a semicolon (;) in C and a right parenthesis in MUMPS.

- Variables from the application program, called host variables, can be used in embedded SQL statements wherever constants are allowed. These can be used on input to tailor an SQL statement to a particular situation and on output to receive the results of a query.
- Queries that return a single row of data are handled with a singleton SELECT statement; this statement specifies both the query and the host variables in which to return data.
- Queries that return multiple rows of data are handled with cursors. A cursor keeps track of the current row within a result set. The DECLARE CURSOR statement defines the query, the OPEN statement begins the query processing, the FETCH statement retrieves successive rows of data, and the CLOSE statement ends query processing.
- While a cursor is open, positioned update and positioned delete statements can be used to update or delete the row currently selected by the cursor.

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms713573(v=vs.85).aspx>

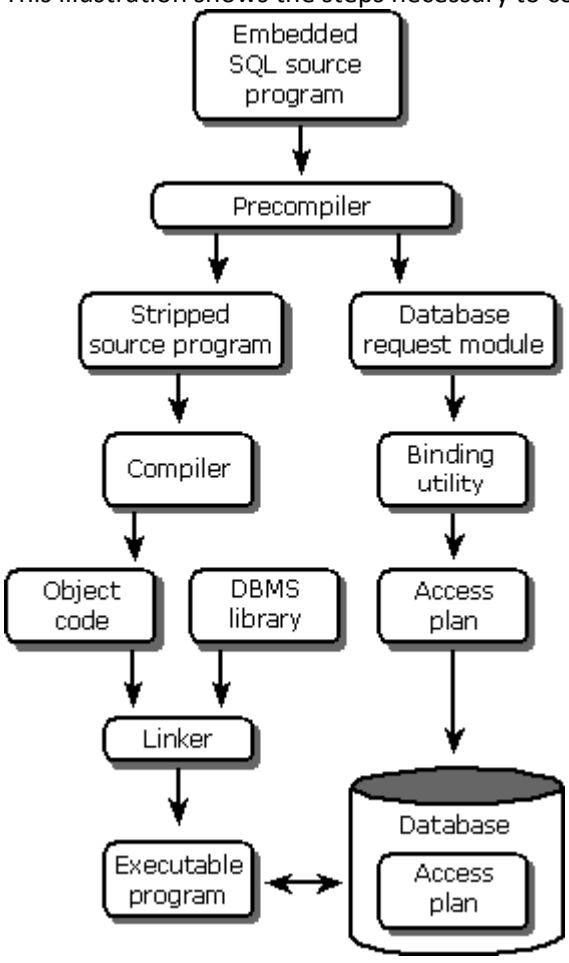
The following code is a simple embedded SQL program, written in C. The program illustrates many, but not all, of the embedded SQL techniques. The program prompts the user for an order number, retrieves the customer number, salesperson, and status of the order, and displays the retrieved information on the screen.

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms714570(v=vs.85).aspx>

```
int main() {
  EXEC SQL INCLUDE SQLCA;
  EXEC SQL BEGIN DECLARE SECTION;
    int OrderID;       /* Employee ID (from user)       */
    int CustID;        /* Retrieved customer ID         */
    char SalesPerson[10]  /* Retrieved salesperson name  */
    char Status[6]       /* Retrieved order status       */
  EXEC SQL END DECLARE SECTION;
/* Set up error processing */
  EXEC SQL WHENEVER SQLERROR GOTO query_error;
  EXEC SQL WHENEVER NOT FOUND GOTO bad_number;
/* Prompt the user for order number */
  printf ("Enter order number: ");
  scanf_s("%d", &OrderID);
/* Execute the SQL query */
  EXEC SQL SELECT CustID, SalesPerson, Status
    FROM Orders
    WHERE OrderID = :OrderID
    INTO :CustID, :SalesPerson, :Status;
/* Display the results */
  printf ("Customer number:  %d\n", CustID);
  printf ("Salesperson: %s\n", SalesPerson);
  printf ("Status: %s\n", Status);
  exit();
query_error:
  printf ("SQL error: %ld\n", sqlca->sqlcode);
  exit();
bad_number:
  printf ("Invalid order number.\n");
  exit();
}
```

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms714570(v=vs.85).aspx>

Because an embedded SQL program contains a mix of SQL and host language statements, it cannot be submitted directly to a compiler for the host language. Instead, it is compiled through a multistep process. Although this process differs from product to product, the steps are roughly the same for all products.
This illustration shows the steps necessary to compile an embedded SQL program.



Five steps are involved in compiling an embedded SQL program:

1. The embedded SQL program is submitted to the SQL precompiler, a programming tool. The precompiler scans the program, finds the embedded SQL statements, and processes them. A different precompiler is required for each programming language supported by the DBMS. DBMS products typically offer precompilers for one or more languages, including C, Pascal, COBOL, Fortran, Ada, PL/I, and various assembly languages.
2. The precompiler produces two output files. The first file is the source file, stripped of its embedded SQL statements. In their place, the precompiler substitutes calls to proprietary DBMS routines that provide the run-time link between the program and the DBMS. Typically, the names and the calling sequences of these routines are known only to the precompiler and the DBMS; they are not a public interface to the DBMS. The second file is a copy of all the embedded SQL statements used in the program. This file is sometimes called a database request module, or DBRM.
3. The source file output from the precompiler is submitted to the standard compiler for the host programming language (such as a C or COBOL compiler). The compiler processes the source code and produces object code as its output. Note that this step has nothing to do with the DBMS or with SQL.
4. The linker accepts the object modules generated by the compiler, links them with various library routines, and produces an executable program. The library routines linked into the executable program include the proprietary DBMS routines described in step 2.
5. The database request module generated by the precompiler is submitted to a special binding utility. This utility examines the SQL statements, parses, validates, and optimizes them, and then produces an access plan for each statement. The result is a combined access plan for the

entire program, representing an executable version of the embedded SQL statements. The binding utility stores the plan in the database, usually assigning it the name of the application program that will use it. Whether this step takes place at compile time or run time depends on the DBMS.

Notice that the steps used to compile an embedded SQL program correlate very closely with the steps described earlier in Processing an SQL Statement. In particular, notice that the precompiler separates the SQL statements from the host language code, and the binding utility parses and validates the SQL statements and creates the access plans. In DBMSs where step 5 takes place at compile time, the first four steps of processing an SQL statement take place at compile time, while the last step (execution) takes place at run time. This has the effect of making query execution in such DBMSs very fast.

From <http://msdn.microsoft.com/en-us/library/windows/desktop/ms713968(v=vs.85).aspx>

Procedurální prostředky SQL (procedurální rozšíření SQL)

- SQL bylo původně navrženo pro získávání dat z relačních databází. SQL je deklarativní jazyk a ne imperativní, jako například C nebo Java.
- Dodavatelům DB systémů možnosti SQL nedostačovaly a tak si začali implementovat vlastní procedurální rozšíření SQL.
- příklad: kursory, bloky, proměnné

Motivace

- Šetření komunikačního kanálu
- Menší množství odesílaných povelů
- v jednom povelu je větší množství příkazů
- Podstatně menší objem přenesených dat - Data se zpracují na serveru bez přenosu na klienta
- Odlehčení klienta - Možnost ukládat a vykonávat kód na serveru

Na co si dát při návrhu pozor

- Hlídat na úrovni databáze všechny manipulace s daty, které ohlídat jdou
  - Cokoli jde zadat uživatelem špatně, bude zadáno špatně
- Integritní omezení, triggery
  - Později je čištění nekonzistentních dat namáhavé a opravy často nemožné
  - Lépe ohlídat vše

Procedurální rozšíření v SQL:1999

SQL:1999 standardizuje procedurální rozšíření. Rozšíření se jmenuje *SQL/PSM - SQL/Persisted Stored Modules*. **Nikdo to moc nedodržuje a všichni si dělají vlastní standardy/implementace**

- funkce a procedury - lze zapsat v SQL i v hostitelském programovacím jazyce
- řídící konstrukce - cykly, větvení, přiřazení

Podpora v databázích

SQL:1999 (SQL/PSM)

- IBM DB2
- MySQL
- PostgreSQL

Vlastní (proprietární) rozšíření

- Oracle (PL/SQL)
- PostgreSQL (PL/pgSQL)
- Microsoft (T-SQL)
- Sybase (T-SQL)

SQL, jazyk PL/SQL.

Thursday, May 30, 2013 8:15 AM

SQL - *Structured Query Language* (Strukturovaný dotazovací jazyk) je standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích

Rozdělení standardního SQL

SQL příkazy se dají rozdělit do několika kategorií podle toho co provádíte

Data definition language (DDL) neboli příkazy určené pro práci se strukturou databázových objektů. Nejčastěji tabulek.

CREATE - vytvoření

ALTER - změně

DROP - odstranění

RENAME - přejmenování

TRUNCATE - smazání, aniž by se data ukládala do koše

COMMENT - přidání komentáře

Vloženo z <http://www.tomas-solar.com/index.php/odborne-dotazy/35-co-znamenaji-zkratky-dml-ddl-dcl.html>

Data manipulation language (DML) neboli příkazy určené pro manipulaci s daty

SELECT - vybrání dat z databáze

INSERT - vložení

UPDATE - úprava nebo také editace či změna

DELETE - smazaní

MERGE - sloučení

Vloženo z <http://www.tomas-solar.com/index.php/odborne-dotazy/35-co-znamenaji-zkratky-dml-ddl-dcl.html>

Řízení transakcí. Jednotlivé příkazy DML můžete slučovat do transakcí, ale nemusíte.

COMMIT - slouží k potvrzení veškerých změn

ROLLBACK - provede rollback veškerých změn

SAVEPOINT - vytvoří časovou značku ke které se můžete vracet.

Vloženo z <http://www.tomas-solar.com/index.php/odborne-dotazy/35-co-znamenaji-zkratky-dml-ddl-dcl.html>

Data control language (DCL) neboli příkazy sloužící k přídání či odebrání opravnění k databázi a objektů v ní.

GRANT - přiřazení

REVOKE = odebrání

Vloženo z <http://www.tomas-solar.com/index.php/odborne-dotazy/35-co-znamenaji-zkratky-dml-ddl-dcl.html>

PL/SQL

PL/SQL přidává k jazyku SQL konstrukce procedurálního programování.

PL/SQL (Procedural Language/Structured Query Language) je procedurální nadstavba jazyka SQL od firmy Oracle založená na programovacím jazyku Ada.
PL/SQL je rozšíření jazyka SQL o procedurální rysy. Je specifické pro produkty firmy Oracle, procedurální rozšíření SQL produktů jiných firem se zpravidla navzájem liší. Výjimkou je ŠRBD DB2 společnosti IBM, který podporuje jak vlastní procedurální jazyk SQL PL, tak je plně kompatibilní s jazykem PL/SQL včetně datových typů. Základním stavebním kamenem PL/SQL je tzv. **PL/SQL blok**, který může být buď tělem triggeru, procedry a funkce, nebo samostatný. Struktura PL/SQL bloku je viz "základní konstrukce"
Vloženo z <http://www.kiv.zcu.cz/~zima/vyuka/db2/cviceni-plsql-zaklady.html>

Základním stavebním kamenem v PL/SQL je **blok**. Program v PL/SQL se skládá z bloků, které mohou být vnořeny jeden do druhého. Obyčejně každý blok spouští jednu logickou akci v programu. Blok má následující strukturu:

Základní konstrukce

```
DECLARE
 deklarace
BEGIN
 výkonná část
EXCEPTION
 ošetrení výjimek
END;
```

```
Komentáře:

-- komentář do konce řádky
/* komentář od do */
```

Pouze výkonná sekce je povinná, ostatní jsou doporučené. Jediné příkazy jazyka SQL, které jsou ve výkonné sekci **povolené, jsou SELECT, INSERT, UPDATE, DELETE** a několik dalších pro manipulaci s daty a pro kontrolu transakcí. **Definiční příkazy jazyka SQL jako CREATE, DROP nebo ALTER nejsou povoleny**. Avšak použití těchto příkazů lze pomocí direktivy EXECUTE IMMEDIATE "PŘÍKAZ". PL/SQL není citlivé na velikost písmen a mohou být použity komentáře ve stylu jazyka C.

Vloženo z <http://cs.wikipedia.org/wiki/PL/SQL>

Kurzor

- Kurzor je pracovní oblast obsahující data (výsledná množina, tzv. result set), které lze dále využívat prostřednictvím operací nad kurzory
- Existují implicitní a explicitní kurzory
  - Implicitní jsou jednořádkové SQL (INTO)
  - Explicitní můžeme deklarovat v části DECLARE pomocí klíčového slova CURSOR
- Práce s kurzory se podobá souborům

Procedury + funkce

Bloky příkazů jazyka PL/SQL lze pojmenovat a uložit ve spustitelné formě do databáze = procedury + funkce

• Jsou uloženy ve zkompilovaném tvaru v databázi.
• Mohou volat další procedury či funkce, či samy sebe.
• Lze je volat ze všech prostředí klienta.

Funkce, na rozdíl od procedury, vrací jedinou hodnotu (procedura může vracet hodnot více, resp. žádnou).

Triggery

- uživatelsky definovaný blok PL/SQL sdružený s určitou tabulkou. Je implicitně spuštěn (proveden), jestliže je nad tabulkou prováděn aktualizační příkaz

Anonymous Blocks (jenom BEGIN až END bez názvu, takže zapomenout EXEC)

Packages

- Programový balík je sdružením řady funkcí a procedur s vlastním jmenným prostorem a vlastním persistentním prostorem pro proměnné v rámci jedné session
- Umožňuje uchovávat hodnoty v rámci session pro řadu procedur a funkcí
- Ne náhodná analogie s objekty

Nested Tables

- Představují pole - množina (set, bag) v některých programovacích jazycích
- Lze ukládat tyto tabulky v databázových tabulkách (oboustranná kompatibilita)
- Deklarace:
  DECLARE TYPE ntable IS TABLE
  OF element_type;
- Typ prvku může být lib. PL/SQL typ mimo odkazu (REF) a kurzoru (CURSOR)

Varrays

- variable-size array (dynamické pole) – tzv. varrays odpovídají klasickým dynamickým polím, uchovávají definovaný počet hodnot,
- pomalejší přístup SQL nástroji než k nestedtables

Loops (FOR/WHILE)

```
LOOP
pocet:= pocet +1
IF pocet =100 THEN EXIT;
END IF;
END LOOP;
```

IF

```
IF podmínka THEN příkazy_1;
ELSIF podmínka THEN příkazy_2;
.
.
.
ELSE příkazy_n;
END IF;
```

CASE

```
CASE proměnná
WHEN výraz_1 THEN příkazy_1;
WHEN výraz_2 THEN příkazy_2;
WHEN výraz_n THEN příkazy_n;
ELSE příkazy_n+1
END CASE
```

Kurzory – definice, klasifikace, použití kurzorů.

Thursday, May 30, 2013 8:15 AM

Definice

- Kurzor je abstraktní datový typ umožňující procházet záznamy vybrané dotazem, který je s kurzorem spojen.
- Prostředek pro získání informace z databáze a předání do programu v jazyce PL/SQL
- SELECT co vrací více řádků, přes kurzor je možné výsledky procházet

Klasifikace

- explicitní kurzor
  - nutno deklarovat, otevřít, načíst data a uzavřít DECLARE CURSOR <jméno kurzoru>IS <dotaz>; OPEN <jméno kurzoru>; FETCH <jméno kurzoru>INTO <jméno proměnné1>, <jméno proměnné2>, …; CLOSE <jméno kurzoru>;
- implicitní kurzor
  - je deklarován a prováděn přímo v těle programu
  - v tomto typu kurzoru jsou povoleny pouze příkazy SQL, které vrací jednotlivé řádky nebo nevrací žádné řádky,
  - příkazy SELECT, UPDATE, INSERT a DELETE obsahují implicitní kurzory
  - musí se shodovat datové typy sloupců a proměnných
  - implicitní kurzor SELECT musí vracet pouze jeden řádek (SELECT INTO !) SELECT <jméno sloupce 1>, <jméno sloupce 2> INTO <jméno proměnné 1>, <jméno proměnné 2> FROM ... ;

Atributy kurzoru

- **cursor%FOUND** obsahuje záznamy?
- **cursor%NOTFOUND** neobsahuje záznamy?
- **cursor%ISOPEN** je otevřený?
- **cursor%ROWCOUNT** dosud zpracováno řádků

Použití kurzorů

Když je potřeba iterovat přes hromadu položek a pro každou položku něco provést (tedy ne pro všechny najednou, ale pro každou zvlášť)

- v triggerech
- v uložených procedurách

příklad:

```
DECLARE
  tmp osoby%ROWTYPE;
  CURSOR plist IS SELECT * FROM osoby;
BEGIN
  OPEN plist;
  LOOP
    FETCH plist INTO tmp;
    EXIT WHEN plist%NOTFOUND;
    dbms_output.put_line(plist%ROWCOUNT||'. '||tmp.jmeno||'
'||tmp.prijmeni);
  END LOOP;
  CLOSE plist;
END;
```

- Příklad implicitního kurzoru:

```
DECLARE
Sum NUMBER;
BEGIN
```

```
    SELECT SUM(salary) INTO Sum FROM Employee
END;
```

Vloženo z <http://www.kasman.sk/kurzor-v-oracle>

- Dále např. automatické číslování v SŘBD Oracle (Autoinkrement)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Jaká konstrukce se používá pro zpracování víceřádkových SELECT v PL/SQL a jaké jsou její dva typy?

Používá se konstrukce CURSOR – předává programu PL/SQL info z databáze
Explicitní kurzor - nutno deklarovat, otevřít, načíst data a uzavřít (vše dělá programátor)
Implicitní kurzor - je automaticky deklarován Oraclem a prováděn přímo v těle programu při použití SQL dotazu. Implicitní je proto, že ho uživatel nemusí nijak deklarovat. **Pokud např. Select vrací více řádku, se kterými chceme dále pracovat, je nutné použít explicitní kurzor.**

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/DB2_zkouska.docx>

Uložené procedury, funkce a balíky procedur a funkcí, kompilace, spouštění.

Thursday, May 30, 2013 8:16 AM

Podprogram je pojmenovaný blok, který může být opakovaně volán a může přebírat aktuální parametry. Typy podprogramů jsou **funkce a procedury**. Procedury a funkce lze sdružovat do logických celků – balíků (package).

Všechny anonymní bloky PL/SQL lze natrvalo uložit do databáze, a to buď ve tvaru procedury, nebo funkce.

Uložené podprogramy – procedury a funkce

Procedury a funkce mohou být trvale uloženy do DB a mohou být použity jakoukoli aplikací, která s databázovým strojem komunikuje. Jednou přeložený a uložený program patří mezi databázové objekty a může být referován libovolným počtem aplikací.

Uložené podprogramy mohou být samostatné, nebo součástí balíku.

Podprogramy lze volat z:

- DB triggerů
- Jiných uložených podprogramů
- Aplikačních programů zapsaných ve vyšším programovacím jazyce, pro nějž existuje předkompilátor (ORACLE Pro)
- Interaktivně (SQL*Plus) : EXECUTE jmeno_procedury(parametry)

Uložené procedury a funkce mohou mít parametry, které mohou být:

- IN - vstupní
- OUT - výstupní
- IN OUT – vstupní i výstupní

Uložená procedura

Procedura se vstupními parametry:

- Nelze volat z SQL příkazů
- Na rozdíl od funkce může vracet více parametrů (IN/OUT)

```
CREATE OR REPLACE PROCEDURE nastav_plat (id IN NUMBER, novy_plat
IN NUMBER) AS
zam_plat NUMBER(5,2);
nema_plat EXCEPTION;
BEGIN
SELECT plat INTO zam_plat FROM osoby WHERE os_cislo = id;
IF zam_plat IS NULL THEN
RAISE nema_plat;
END IF;
EXCEPTION
WHEN nema_plat THEN
UPDATE osoby SET plat = novy_plat WHERE os_cislo = id;
COMMIT;
END;
```

Pro zrušení (smazání) uložené procedury použijeme příkaz:

```
DROP PROCEDURE nastav_plat;
```

Uložená funkce

Způsob vytvoření uložené funmkce (s parametry) ukazuje následující příklad:

```
CREATE OR REPLACE FUNCTION secti(a IN INTEGER, b IN INTEGER)
RETURN INTEGER AS
```

```
BEGIN
RETURN (a + b);
END;
Pro zrušení (smazání) uložené funkce použijeme příkaz:
DROP FUNCTION secti;
```

Uložené balíky

Uložené balíky procedur a funkcí slouží ke sdružení logicky spolu souvisejících procedur a funkcí, ale i typů a objektů. Mohou obsahovat i globální proměnné, jejichž platnost je omezena délkou aktuálního spojení s databází.

Definice balíku představuje definici rozhraní balíku (deklarace typů, proměnných, konstant, podmínek definujících nestandardní stavy, kurzorů, podprogramů dostupných zvenčí)(co tam je za funkce parametry, ale bez těla) a těla balíku (implementuje specifikaci)(normálně napsané procedury včetně implementace).

Př. rozhraní:

```
CREATE OR REPLACE PACKAGE arithmetic AS
usage INTEGER;

FUNCTION add(a IN INTEGER, b IN INTEGER) RETURN INTEGER;
FUNCTION sub(a IN INTEGER, b IN INTEGER) RETURN INTEGER;
PROCEDURE inc(a IN OUT INTEGER);
END;
/
```

Př. Těla:

```
CREATE OR REPLACE PACKAGE BODY arithmetic AS

FUNCTION add(a IN INTEGER, b IN INTEGER) RETURN INTEGER IS
BEGIN
usage := usage + 1;
RETURN (a + b);
END;

FUNCTION sub(a IN INTEGER, b IN INTEGER) RETURN INTEGER IS
BEGIN
usage := usage + 1;
RETURN (a - b);
END;

PROCEDURE inc(a IN OUT INTEGER) IS
BEGIN
usage := usage + 1;
a := a + 1;
END;
BEGIN
usage := 0;
END;
/
```

Kompilace

- Pokud je volána procedura/funkce ve stavu INVALID, kompiluje se automaticky.
  - Pokud se kompilace nezdaří, dojde k výjimce.
- Ruční kompilace

- `ALTER PROCEDURE[FUNCTION] jmproc COMPILE;`
  - Pokud se kompilace nezdaří, dojde k výjimce
- SQL*Plus a chyby kompilace – pomocí něj můžeme zjistit, jaké chyby se vyskytly při kompilaci. Pokud uložení procedury/funkce neproběhlo bez chyb, nelze ji používat a je nutné ji opravit. Pomocí SQL*Plus příkazů:
  - SHOW ERROR – vypíše popis poslední chyby, na kterou při ukládání (kompilaci) narazil

- SHOW ERR typ jméno – např. SHOW ERR FUNCTION F – pro uvedený objekt
- Pomocí dotazu na tabulku USER_ERRORS

Spouštění

Proceduru můžeme zavolat (spustit různými způsoby:

- Pomocí direktivy EXEC – EXEC nastav_plat(123, 10000);
- V těle jiného PL/SQL bloku

```
BEGIN
...
Nastav_plat(123, 10000);
...
END;
/
```

Funkci můžeme volat také dvěma způsoby:

- V příkazu SELECT – SELECT secti(2, 3) FROM dual;
- V těle jiného PL/SQL bloku

```
DECLARE
a INTEGER;
b INTEGER;
BEGIN
a := 5;
b := secti(a, 2);
dbms_output.put_line('Vysledek : '||b);
END;
/
```

Při volání funkcí/procedur z balíků používáme tečkovou notaci (package.funkce()) pro kvalifikaci jejich jména.

Zabalený podprogram lze volat z db triggeru, jiného uloženého programu, aplikace napsané pro některý z předkompilátorů, standardních klientských nástrojů (SQL*Plus)

Standardní balík ***STANDARD*** = definuje prostředí PL/SQL.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Aktivní databáze – Oracle triggery, klasifikace a spouštění triggerů.

Thursday, May 30, 2013 8:16 AM

V řadě IS, které běží nad nějakou databází, potřebujeme v případě vzniku nějaké události (např. modifikujeme řádek v nějaké tabulce) automaticky spustit příkaz, který provede nějaké operace. K tomuto účelu slouží triggery (spouštěče). Trigger je speciální typ uložené procedury, která se aktivuje při splnění nějaké podmínky na serveru (aktualizace dat, události spojené s DB nebo session).

Triggery jsou vlastně procedury, které automaticky volá (spouští) SŘBD při definované události. Touto událostí může být buď vložení (INSERT), rušení (DELETE), nebo aktualizace (UPDATE OF) záznamu v tabulce (lze vázat i na update konkrétního sloupce).Pravidla lze kombinovat pomocí OR. Triggery bývají obvykle volány buď:

- Před specifikovanou událostí – **BEFORE**
- Po specifikované události – **AFTER**
- **INSTEAD OF** - místo specifikované události

Triggery jsou:

- Spuštěny
- Vyhodnoceny
- Vykonány

Od uložených procedur a funkcí se odlišují tím, že jsou spuštěny *při modifikaci tabulky*, definují se pouze pro db tabulky a nepřijímají argumenty a lze je spustit jen při zmiňovaných DML příkazech.

U triggeru lze rovněž specifikovat podmínku, kdy má být vykonáno jeho tělo (PL/SQL blok) a to použitím klauzule WHEN.

Triggery jsou plně zkompilované po spuštění příkazem CREATE TRIGGER a po uložení procedurálního kódu v systémovém katalogu.

Trigger lze také

- deaktivovat (zakázat) - ALTER TRIGGER jmeno DISABLE;
- aktivovat – ALTER TRIGGER jmeno ENABLE;
- zrušit – DROP TRIGGER jmeno;

Výhody triggerů jsou:

- nepovolí neplatné datové transakce, zajišťují komplexní bezpečnost, zajišťují referenční integritu přes všechny uzly db, zajišťují audit (sledování), spravují synchronizaci tabulek, zaznamenávají statistiku často modifikovaných tabulek.

Klasifikace triggerů

Příkazové triggery (statement level)

Triggery se spustí nad tabulkou bez ohledu na to, kolik tabulka obsahuje řádek. Např. pokud chci logovat změny provedené v DB, která obsahuje tabulky PRACOVISTE, ZAMESTNANCI do tabulky LOGY. Po každé operaci I, U, D nad tabulkami PRAC a ZAM se do tabulky LOGY vloží záznam o modifikaci tabulky a typu modifikace.

```
CREATE TRIGGER tai_pracoviste
AFTER INSERT ON pracoviste
BEGIN
INSERT INTO logy VALUES ( 'PRACOVISTE', 'I');
END;
/
```

Řádkové triggery – FOR EACH ROW

Trigger se spouští jednou pro každý řádek tabulky. Např. z předchozího příkladu chci mít u každého záznamu informaci, kdy byl zadán a kdo jej zadal.

```
CREATE TRIGGER trbi_pracoviste
BEFORE INSERT ON pracoviste
FOR EACH ROW
BEGIN
:new.zadal := user;
:new.datum := sysdate;
END;
/
```

Vlastní obsluhu události lze nadefinovat v PL/SQL bloku. Uvnitř PL/SQL bloku (a také klauzuli WHEN) řádkového triggeru se lze odkazovat na původní a nový záznam pomocí pseudoproměnných **:new** (obsahuje vkládaný záznam) a **:old** (původní záznam). Je zřejmé, že při vkládání nového záznamu není definována :old a při mazání :new. Jejich jména lze předefinovat v klauzuli **REFERENCINGOLD AS.**

Business rules triggery

Triggery jsou také často používány při realizaci tzv. business rules, tj. integritních omezení specifických pro danou oblast použití. Např. použijeme – studenti si mohou zapsat max 20 kreditů za semestr. Pokud při vkládání dat do tabulky ZAPIS překročíme maximální povolenou hodnotu, dostaneme chybové hlášení – a to zařídí business trigger.

Zápis

CREATE OR REPLACE TRIGGER jméno
BEFORE | **AFTER** | INSTEAD OF
DELETE | **INSERT** | **UPDATE** OF cols
ON tabulka
[způsob odkazování]
[**FOR EACH ROW**]
[WHEN (podmínka)]
AS pl/sql kód

Omezení triggerů

- BEFORE a AFTER triggery nelze specifikovat nad pohledy
- V BEFORE triggerech není možné zapisovat do :old záznamů
- V AFTER triggerech nelze zapisovat do :old ani do :new záznamů
- INSTEAD OF triggery pracují jen s pohledy, mohou číst :old i :new, ale nemohou zapisovat ani do jednoho
- ~~Nelze kombinovat INSTEAD OF a UPDATE~~
- Nelze definovat trigger nad LOB atributem

Dvě zásadní omezení

- Nelze použít transakce, pokud je zpracovávána jiná transakce, tedy prakticky nelze použít transakce vůbec
- Není možné sledovat ani modifikovat data v tabulce, která způsobila vyvolání DML triggeru -> jediné známé řešení je zrcadlení tabulek

Sémantika Oracle Triggerů

- Spouštění probíhá okamžitě při události, nelze je spustit explicitně (např. uživ. Příkazem)
- Vnořené spouštění triggerů - činnost triggeru může vyvolat jiný trigger - kontext aktuálního triggeru se uloží, začne se vykonávat nový trigger, pak se zas obnoví a pokračuje ten původní
- Maximální hloubka zanoření je 32, pak to hodí výjimku

Doplňující pojmy

události (events)

◦ změna stavu databáze

◦ časové události

◦ externí, definované aplikací

podmínky (conditions)

◦ databázový predikát

◦ databázový dotaz

akce (actions)

◦ libovolná manipulace s daty

- transakční příkazy
- pravidla zpracování
- externí procedury

Spouštění triggerů

- Vykonání triggeru
  - Okamžité (immediate)
    - Před událostí
    - Po události
    - Namísto události
  - Odložené (deferred)
    - Na konci transakce
    - Po uživatelském příkazu
    - Následkem uživatelského příkazu
  - Oddělené (detached)
    - V kontextu samostatné transakce vypuštěné z počáteční transakce poté, co nastala událost
    - Možné kauzální závislosti počáteční a oddělené transakce
- Vykonání akce
  - Okamžité (immediate)
    - Následuje ihned po vyhodnocení podmínky
  - Odložené (deferred)
    - Akce je odsunuta na konec transakce
    - Akci vyvolá uživatelský příkaz
  - Oddělené (detached)
    - Probíhá v kontextu samostatné transakce vypuštěné z počáteční transakce ihned po vyhodnocení podmínky
    - Možné kauzální závislosti počáteční a oddělené transakce

Kdy je co spuštěno

BEFORE triggers run the trigger action before the triggering statement is run. This type of trigger is

commonly used in the following situations:

When the trigger action determines whether the triggering statement should be allowed to complete. Using a BEFORE trigger for this purpose, you can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action.
To derive specific column values before completing a triggering INSERT or UPDATE statement.
[7:01:26 PM] Lukáš Volf: AFTER triggers run the trigger action after the triggering statement is run.
[7:01:50 PM] Lukáš Volf: INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through DML statements (INSERT, UPDATE, and DELETE). These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.

You can write normal INSERT, UPDATE, and DELETE statements against the view and the INSTEAD OF trigger is fired to update the underlying tables appropriately. INSTEAD OF triggers are activated for each row of the view that gets modified.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Transakce, dvoufázový uzamykací protokol, detekce uváznutí.

Thursday, May 30, 2013 8:17 AM

Transakce

Databázové transakce musí splňovat tzv. vlastnosti ***ACID***

- *A - Atomicity*
  Databázová transakce je jako operace dále nedělitelná (atomická). Provede se buď jako celek, nebo se neprovede vůbec (a daný databázový systém to dá uživateli na vědomí, např. chybovou hláškou).
- *C - Consistency - konzistentnost*
  Při a po provedení transakce není porušeno žádné integritní omezení.
- *I - Isolation - izolovanost*
  Operace uvnitř transakce jsou skryty před vnějšími operacemi. Vrácením transakce (ROLLBACK) není zasažena jiná transakce, jinak i tato musí být vrácena. V důsledku tohoto chování může dojít k tzv. řetězovému vrácení (cascading rollback).
- *D - Durability - trvalost*
  Změny, které se provedou jako výsledek úspěšných transakcí, jsou skutečně uloženy v databázi a již nemohou být ztraceny.

Globální vs. lokální

- Lokální transakce probíhá pouze na jediném uzlu.
- Globální (distribuovaná) transakce přesahuje rozsah jednoho uzlu.

Stavy transakce

- Aktivní - od počátku provádění transakce
- Částečně potvrzený - stav po provedení poslední operace transakce
- Chybný - nelze pokračovat v normálním průběhu transakce
- Zrušený - nastane po skončení operace ROLLBACK
- Potvrzený - po úspěšném vykonání COMMIT

Prováděné operace

Pro práci s transakcemi je nutné zavést následující operace:

- BEGIN - začátek transakce
- COMMIT - ukončení transakce a uložení dosažených výsledků do databáze
- ROLLBACK - odvolání změn - není-li definován savepoint, (místo, po které lze provedené změny vrátit zpět) tak návrat do stavu před započetím vykonávání transakce

Optimistické vs. pesimistické zamykání

- U ***pesimistického zpracování*** se v jeho průběhu změny zaznamenávají do dočasných objektů (například a nejčastěji: do řádků tabulek s příznakem dočasných dat, platných jen po dobu transakce) a teprve po přesunu/změně dat se odznačí příznak dočasnosti a data se stanou platnými. (Tento způsob se dá přibližně připodobnit přepisu souboru, při kterém se nejdříve nová verze souboru nakopíruje pod dočasným jménem a teprve poté se tento soubor přejmenuje za starý a tím ho nahradí.)
- U ***optimistického zpracování*** se (optimisticky) předpokládá, že při transakci nenastane chyba a nebude třeba ji vrátit zpět (přestože tato možnost je zachována). Měněné záznamy v tabulkách jsou při optimistickém zpracování transakce zapisovány „natvrdo“, současně s tím se však vytváří tzv. rollback log coby seznam SQL příkazů, které dokáží prováděné změny vrátit zpět. V případě, že při transkaci dojde k nějaké nezotavitelné chybě, tento log se provede a transakce (aby dodržela pravidlo atomicity) skončí ve výchozím stavu s chybou. Naopak, na konci transakce, při které k žádné takové chybě nedošlo, se rollback log maže.

Žurnály

Jsou záznamy, které uchovávají informace o průběhu transakcí a slouží k zotavení po vzniklé chybě. Žurnály musí být v každém uzlu a obsahují záznamy o historii každé transakce.

Dvoufázový protokol (2PL)

Dvoufázová transakce v první fázi zamyká vše co je potřeba a od prvního odemknutí (druhá fáze) již jen odemyká co měla zamčeno (již žádná operace LOCK). Tedy transakce musí mít všechny objekty uzamčeny předtím, než nějaký objekt odemkne. Dá se dokázat, že pokud jsou všechny transakce v dané množině transakcí dobře formované a dvoufázové, pak každý jejich legální rozvrh je uspořádatelný. Dvoufázový protokol zajišťuje uspořádatelnost, ale ne zotavitelnost ani bezpečnost proti kaskádovému rušení transakcí nebo uváznutí.

Striktní dvoufázový protokol (S2PL)

Problémy 2PL jsou nezotavitelnost a kaskádové rušení transakcí. Tyto nedostatky lze odstranit pomocí striktních dvoufázových protokolů, které uvolňují zámky až po skončení transakce (COMMIT). Zřejmá nevýhoda je omezení paralelismu. 2PL navíc stále nevylučuje možnost deadlocku. Read-lock je tu možné uvolnit kdykoli během 2. fáze, write-lock jen na konci.

SS2PL

Prakticky se dnes používá SS2PL, což je Strong Strict twophase locking, které sice nevylučuje deadlock, ale pokud k němu dojde, tak ho umí automaticky vyřešit. Liší se od SS2PL jen tím, že i read-lock uvolňuje až na konci.

Konzervativní dvoufázový protokol (C2PL)

Rozdíl oproti 2PL je ten, že transakce žádá o všechny své zámky, ještě než se začne vykonávat. To sice vede občas k zbytečnému zamykání (nevíme co přesně budeme potřebovat, tak radši zamkneme víc), ale stačí to již k prevenci uváznutí (deadlocku). Pomalý, je třeba vědět předem, co se bude číst/zapisovat, nepoužívá se.

Detekce uváznutí a zotavení

- Uváznutí se detekuje pomocí čekacího grafu
  - Vrcholy jsou transakce Ti
  - Orientovaná hrana Ti → Tj značí, že Ti čeká, až Tj odemkne datovou položku
  - Je-li v čekacím grafu cyklus, došlo k uváznutí
- Hledá se takový plán transakcí, aby se co nejmíň kryly a tak, aby byl dodržen princip ACID (hlavně Isolation), když se takový plán povede najít, nazývá se uspořadatelný
- ***Well formed transakce*** (správně zamykat a odemykat)
- ***Když se zjistí uváznutí***
  - Je nutno nalézt obětní transakci a vnutit jí abort (a tím i obnovu dat). Obětuje se obvykle nejmladší transakce, tj. ta, která ještě neudělala mnoho změn
  - Transakce mohou stárnout, bude-li za oběť vybírána vždy nejmladší transakce. Proto je vhodné do kritéria výběru obětí zahrnout i počet transakcí provedených návratů.
  - Která data se ale mají obnovovat?
    - ***Totální obnova*** transakci úplně zruší, data se vrátí do počátečního stavu, a transakce se restartuje. To může být velmi nákladné
    - Efektivnější je, když se transakce "vrací postupně" do stavu, kdy uváznutí zmizí. Tento postup je ale náročný na evidenci kroků a změn transakcí provedených: metoda kontrolních bodů (***checkpointing***) – konzistentní mezistavy

Zajištění uspořadatelnosti pomocí pořadových čísel transakcí

Transakce vyvolá write W(x):

1. TSR(x) > TS(t): zápis do „později přečtené“ paměti > ROLLBACK
2. TSW(x) > TS(t): zápis do „později přepsané“ paměti > ROLLBACK
3. jinak proveď zápis

Transakce vyvolá read R(x):

4. TSW(x) > TS(t): čtení z „později zapsané“ paměti > ROLLBACK
5. jinak read

Využití časových razítek - time stamp viz stará přednáška 5 nebo link:
http://www.inf.fu-berlin.de/lehre/WS03/DBSII/unterlagen/dbsII-03-17-DDBCC1-6.pdf

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Optimalizace dotazu, jednotlivé přístupy (např. Cost Based optimalizace (CBO)), podstata optimalizátoru, přínos optimalizace.

Thursday, May 30, 2013 8:17 AM

- SQL velmi flexibilní – dvěma i více různými dotazy je možné obdržet stejná data, ovšem rychlost dotazů nemusí být stejná
- důvodem optimalizace je minimalizace nákladů na:
  - strojový čas
  - kapacitu paměti či prostoru
  - programátorskou práci
  - přenesená data
- u malých databází optimalizace nepatrná, projeví se až u objemných, nebo u často navštěvovaných webů může při špatně formulovaných dotazech vzrůst trafic v obou směrech

zpracování příkazů se skládá z následujících komponent:

- parser
- optimalizátor
- generátor řádkových zdrojů
- vlastní provádění SQL

SŘBD Oracle již sám používá optimalizačních technik pro vyhodnocení jakéhokoli dotazu. Těmito technikami jsou:

- Rule based optimaliaztion (RBO)
- Cost based optimalization (CBO) – od verze Oracle 9i je preferovaný

Jak zjistit způsob provedení příkazu? Abychom mohli zjistit, jak ve skutečnosti daná optimalizace vyhodnocení dotazu funguje, potřebujeme vytvořit tabulku **PLAN_TABLE** (podle skriptu *utlxplan.sql),* kam optimalizátor ukládá své vítězné plány právě vyhodnoceného dotazu. Pro vysvětlení (tj. zjištění optimálního plánu) vyhodnocení dotazu použijeme příkaz **EXPLAIN_PLAN.**

```
explain plan for select p.NAZEV,m.ZKRATKA, ...
```

Vykonáním tohoto příkazu se vítězný plán uloží do tabulky PLAN_TABLE v podobě několika záznamů. Informace vyčteme s použitím dotazu:

```
select plan_table_output from table(dbms_xplan.display());
```

CBO/RBO

Oracle doporučuje používat pouze CBO, který je stále vylepšován a RBO je implementován hlavně kvůli zpětné kompatibilitě.

RBO (Rule Based Optimization)

- Starší přístup, dnes často deprecated (Oracle),
- Odvozuje plán ze syntaxe příkazu a existence indexů
  řídí se předem sestavenou sadou pravidel, která nezohledňují např:
  - velikost tabulky -- ***Malá tabulka*** (obsahuje 5 řádků a vejde se do jednoho datového bloku), je rychlejší jí přečíst celou než hledat podle indexu (1 IO operace vs. čtení bloku indexů a pak dat)
  - Možnost špatného výběru použitého indexu - Pokud existuje více neunikátních indexů na jedné tabulce, nemusí optimalizátor vybrat ten nejlepší. Použití určitého indexu je možné optimalizátoru znemožnit použitím výrazu v dotazu.

Ceny přístupu k podmnožině řádek v tabulce v klesajícím pořadí:

- *Plný přístup (Full scan)* – prochází se celá tabulka – všechny záznamy, u každé řádky se ověří podmínka. Vhodné, pokud procento vyhovujících řádek bude velké.
- *Index-Range-Scan* – vyhledání intervalu v indexu. Ověření ostatních podmínek v odkazovaných řádcích
- *Unique-Index-Scan* – vyhledání jediné možné vyhovující řádky podle unikátního indexu
- *ROWID-Scan* – vyhledání řádky na základě známé hodnoty jejího fyzického identifikátoru v DB.

- Z důvodu kompatibility je možné jej však aktivovat odpovídajícím hintem RULE.

CBO (Cost Based Optimization)

- Hledá plán s nejmenšími náklady pomocí statistik. Optimalizace je založena na vyhodnocení kvantitativního využívání zdrojů v průběhu zpracování dotazu (CPU, paměť, I/O,…). Využívá **statistiky** o tabulkách a datech, které jsou uloženy v Data Dictionary:

- Počet různých hodnot ve sloupci, histogramy rozložení hodnot ve sloupci, počet řádek v tabulce, průměrná délka jedné řádky.

- Některé statistiky jsou přístupné i pro uživatele db pomocí pohledů, či tabulek

- Aktualizují se výpočtem nebo odhadem

- **Typy statistik**:

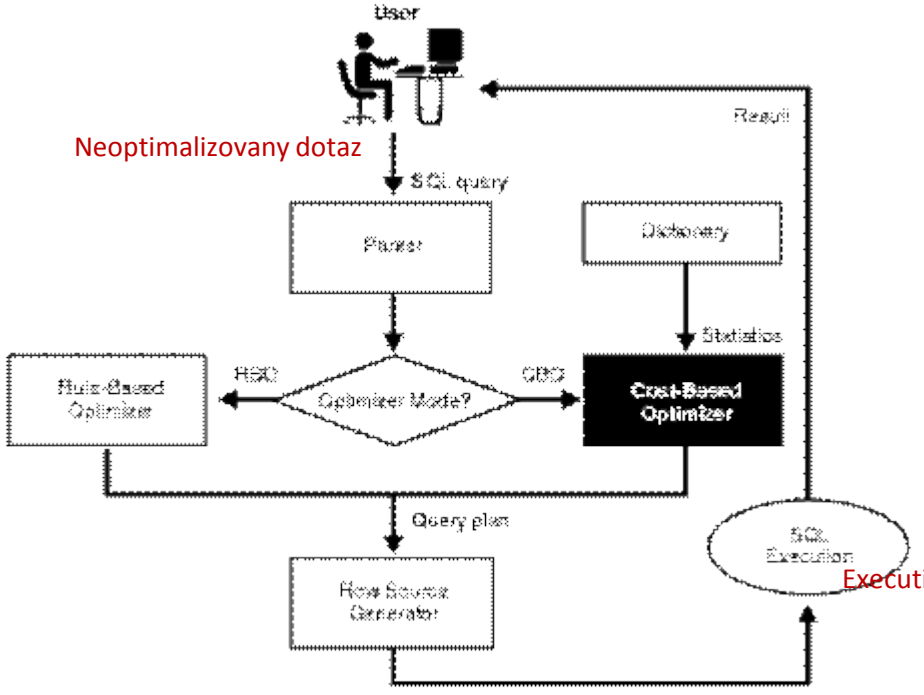
  - **Údaje o tabulkách** – počet řádků, bloků, délka záznamu
  - **Údaje o sloupcích** – počet unikátních hodnot a NULL, histogram
  - **Údaje o indexech** – počet listových bloků, clustering

- Dokáže rozlišit plány i pro různé typy konstant v dotazu.

- Použití CBO je doporučeno firmou Oracle. Vybírá se plán s nejnižší váženou cenou.

Podstata optimalizátoru a přínos

Podstata: stejná data lze z databáze získat různými dotazy (SELECT * vs. SELECT col1, col2…), výsledek bude stejný, ovšem zpracování se může lišit potřebným časem a systémovými nároky.



Execution plan - stromova struktura

Výstupem optimalizátoru je plán vykonávání (execution plan), který určije:

- Přístupové cestu k jednotlivým tabulkám používaných dotazem a pořadí jejich spojování (JOIN order)

Hints

Hint = podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu. Hinty se aplikují na blok dotazu, ve kterém se vyskytují.

V CBO lze využít pro optimalizaci i nápovědu – Hints. Prostřednictvím ní mohu optimalizátoru vnutit některou operaci, protože si myslím, že její užití přispěje k lepší optimalizaci. Tato nápověda se zapisuje jako komentář specifického tvaru.

```
explain plan for select /*+ INDEX (predmety predmety_index1) */
p.NAZEV,…
```

Indexy

- Tvorba indexů není v SQL-92 standardizována
- Jednotlivé databázové systémy řeší tvorbu indexů svými prostředky, které jsou navzájem více či méně podobné
  - Může se lišit syntaxe, podpora různých typů indexů, jejich použití/nepoužití pro daný dotaz
- **B-tree indexy**
  - Obvykle redundantní B+ stromy
    - Hodnoty v listech
    - Listy oboustranně linkované pro snadný sekvenční průchod
    - Vhodné pro sloupce s vysokou selektivitou (počtem různých hodnot ve sloupci)
    - Vícesloupcové (složené) indexy mohou zvýšit selektivitu
    - Nad jednou tabulkou v jednom dotazu nelze obvykle kombinovat více B-tree indexů. Dotaz se vyhodnocuje s použitím jednoho z indexů a ostatní podmínky se dopočítávají.
- **Bitmapové indexy**
  - Pro každou hodnotu sloupce/výrazu vytvořen binární řetězec obsahující 1 právě pro řádky s danou hodnotou
    - Vhodné pro sloupce s nízkou selektivitou
    - Lze kombinovat více bitmapových indexů nad jednou tabulkou pro zvýšení selektivity
    - Kombinací více bitmap se zvyšuje selektivita indexu
- Indexy nepomohou
  - Pokud je procento vyhovujících záznamů velké (zvýšená režie s přístupem k řádkům v nesekvenčním pořadí daném indexem
  - Při dotazech na hodnotu null
    - V indexech se běžně neukládá
- Indexy pomohou
  - V dotazech na rovnost sloupce s konstantou
  - V dotazech na to, zda je hodnota v intervalu
- Indexy jsou automaticky vytvářeny
  - Pro primární klíče
  - Pro sloupce s UNIQUE (kandidátní klíče)
- **Vždy vytvářet indexy pro cizí klíče!!!**
  - Zrychlení odezvy při manipulaci s nadřízenou tabulkou
  - Průchod přes index najde efektivně všechny existující závislé řádky bez nutnosti čtení celé tabulky

Výběr typu optimalizace

Je dalším krokem, kterým můžeme ovlivnit optimalizaci. Jedná se o změnu optimalizačního typu optimalizátoru SŘBD Oracle. Typy optimalizace jsou:

- CHOOSE – výběr podle (ne)přítomnosti statistik nejsou-li k dispozici, potom RBO, jinak CBO.
- ALL_ROWS – vždy CBO, minimalizuje se cena za získání všech řádek odpovědi. Vhodné pro dávkové zpracování.
- FIRST_ROWS – vždy CBO, minimalizuje se cena za získání prvních řádek odpovědi. Vhodné pro interaktivní zpracování.

- RULE – vždy RBO.

Typ optimalizace se vybírá příkazem:

```
ALTER SESSION SET OPTIMIZER_GOAL = ALL_ROWS;
```

Další možnost ladění je změna módu optimalizátoru. Dotazy mohou být optimalizovány na:

- Nejlepší průchodnost – ALL_ROWS
- Nejrychlejší odezvu – FIRST_ROWS_1 – zkrátí čas vyhodnocení dotazu.

Př. nastavení módu:

```
ALTER SESSION SET optimizer_mode = all_rows;
```

Obecná pravidla pro psaní SQL dotazů

Vyplývají z technik optimalizace:

- V selectu nepoužívat v seznamu sloupců *, protože ve většině případů nepracujeme se všemi
- Používat co nejméně klauzuli LIKE, IN, NOT IN (vhodnější je WHERE a WHERE NOT EXISTS)
- Používat klauzule typu LIMIT
- Používat hinty (podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu)
- Na začátek dávat obecnější podmínky (takové, po kterých vypadne co nejvíc záznamů)
- Výběr vhodného pořadí spojení
- Nastavit indexy

Přínos:

- zdrojový čas
- kapacitu paměti či prostoru
- programátorskou práci
- přenesená data

Postrelační databáze – výhody a nevýhody, mapování, RDB, ORDB, OODB.

Thursday, May 30, 2013 8:18 AM

Postrelační databázový systém je relační databázový systém rozšířený o nějakou specializaci na databázové úrovni, jelikož aplikační řešení by bylo nedostačující

Příklady postrelačních DB systémů

- *Prostorové databáze* — rozšířeny o práci s prostorovými objekty a vztahy mezi nimi
- *Objektově orientované databáze* — rozšířeny o objektový model dat a vazby
- *Deduktivní databáze* — rozšířeny o funkce pro analýzu dat
- *Temporální databáze* — rozšířeny o temporální logiku
- *Multimediální databáze* — rozšířeny o funkce pro práci s multimediálním obsahem
- *Aktivní databáze* — rozšířeny o aktivní pravidla

V současnosti všechny používané databázové systémy jsou postrelační, jelikož obsahují nějaká rozšíření oproti původnímu relačnímu schématu (např. triggery).

Vloženo z <http://wp.soulwasted.net/msz/pdb/definice-postrelacniho-db-systemu>

RDB (RSŘBD) - relational database

ORDB (ORSŘBD) - object-relational database

OODB (OOSŘBD) - object oriented database

Jedná se o všechny současné databáze - jde o relační databáze doplněné o nějakou "funkci" navíc, např. aktivní databáze (triggery) už jsou postrelační databáze.

Relační databáze

Technologie relačních databází byla původně navržena E.F.Coddem a později ji implementovala IBM a jiní. Standard je popsán ANSI a ISO normou, častěji se na ni ovšem odvoláváme jako na SQL + číslo verze. Poslední je tedy SQL2. Novější verze SQL3 obsahuje navíc některá objektová rozšíření.

- Datový model

  RDB uchovává data v databázi skládající se z řádků a sloupců. Řádek odpovídá záznamu (record, tuple); sloupce odpovídají atributům (polím v záznamu). Každý sloupec má určen datový typ. Datových typů je omezené množství, typicky 6 nebo víc (např. znak, řetězec, datum, číslo...). Každý atribut (pole) záznamu může uchovávat jedinou hodnotu. Vztahy nejsou explicitní, ale spíše plynou z hodnot ve speciálních polích, tzv. cizí klíče (foreign keys) v jedné tabulce, který se rovná hodnotám v jiné tabulce.

- Dotazovací jazyk

  Pohled (view) je podmnožina databáze, která je výsledkem vyhodnocení dotazu. V RDB je pohled tabulka. RDB využívá SQL pro definici dat, řízení dat a přístupu a získávání dat. Data jsou získávána na základě hodnoty v určitém poli záznamu.

- Výpočetní model

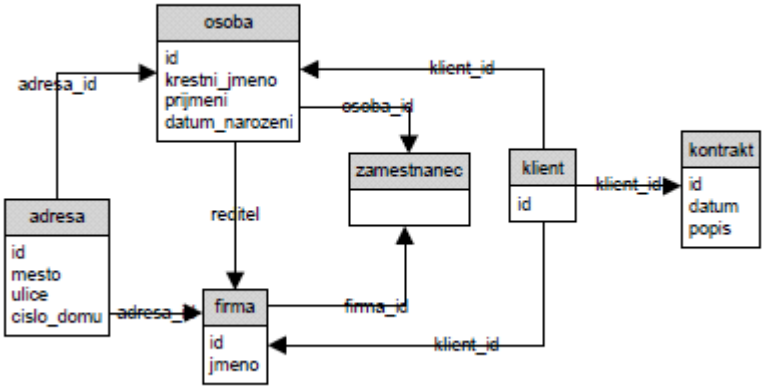
  Veškeré zpracováváni je založeno na hodnotách polí záznamů. Záznamy nemají jednotné identifikátory, které jsou neměnné po dobu existence záznamu. Neexistují žádné odkazy z jednoho záznamu na jiný. Vytvoření výsledku je prováděno pod kontrolou kurzoru, který umožňuje uživateli sekvenčně procházet výsledek po jednotlivých záznamech. Totéž platí pro update.

  Výhody:

  - Výkonné OLTP

- Dostupnost dat
- Utajení
- Prostředky pro správu dat
- Standardní jazykové rozhraní
- Řízení paměti
- Souběžné zpracování dat
- Integrita

Příklad:



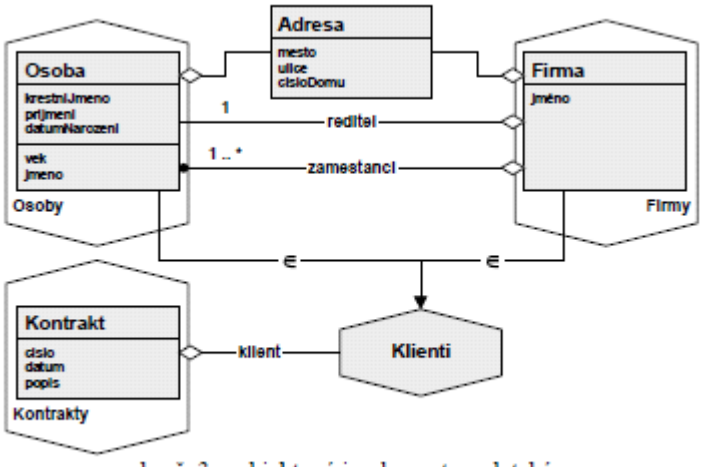
obr. č. 2. - relační implementace databáze

Objektová databáze

Objektově orientované databáze

Vedle relačních databází lidé se začal vyvíjet nový typ databázových systémů, založených na principech objektového programování. Co nového objektové databáze přináší? Tak jako jsme mohli vnímat přechod od strukturálního programování k objektovému programování (např. klasický Turbo Pascal a Delphi), tak můžeme vnímat i přechod z relačních databází na objektové databáze. Základem OO databáze není tentokrát tabulka, ale objekt. Každý objekt má atributy/vlastnosti (zde je vidět analogie se sloupci v tabulce) a metody, které nějakým způsobem manipulují s hodnotami vlastností. Jednotlivé "záznamy" jsou instance objektu s konkrétními hodnotami (v relačních databázích - 1 řádek). Lze zde využít všech výhod dědičnosti (a to i mnohonásobné), zapouzdřenosti a polymorfismu. Díky tomu OO databáze výrazně rozšiřují možnosti tvorby databázových aplikací.

Příklad oproti relační:



obr. č. 3. - objektová implementace databáze

Pro objektové databáze neexistuje žádný oficiální standard. Standardem je de facto kniha Morgana Kaufmana The Object Database Standard: ODMG-V2.0. Důraz ODB je na přímou korespondenci mezi následujícími:

* Objekty a objektové vztahy v aplikaci napsané v OO jazycích

* jejich uchovávání v databázi.

- Objektově orientované databáze (OODB) využívají objektových principů jako jsou abstraktní datové typy, zapouzdření, inheritance, polymorphismus apod. Struktura objektu je (i, c, v) = (unique id, constructor, stav objektu). Unique Object identifiers, …

  OQL = Object Query Language = deklarativní jazyk, přidaná flexibilita

  http://goldberg.berkeley.edu/courses/F04/215/215-OODB.ppt

- Datový model

  Objektové databáze využívají datového modelu, který má objektově orientované aspekty jako třídy s atributy a metodami a integritními omezeními; poskytují objektové identifikátory (OID) pro každou trvalou instanci třídy; podporují zapouzdření (encapsulation); násobnou dědičnost (multiple inheritance) a podporují abstraktní datové typy.

  Objektové databáze kombinují prvky objektově orientovaného programování s databázovými schopnostmi. Rozšiřují funkčnost objektových programovacích jazyků (C++, Smalltalk, Java) a poskytují plnou schopnost programování databáze. Datový model aplikace a datový model databáze se ve výsledku hodně shodují a výsledný kód se dá mnohem efektivněji udržovat.

- Dotazovací jazyk

  Objektově orientovaný jazyk (C++, Java, Smalltalk) je jazykem jak pro aplikaci, tak i pro databázi. Poskytuje těsný vztah mezi objektem aplikace a uloženým objektem. Názorně je to vidět v definici a manipulaci s daty a v dotazech.

- Výpočetní model

  V RDB rozumíme dotazovacím jazykem vytváření, přístup a aktualizaci objektů, ale v ODB, ačkoliv je to stále možné, je toto prováděno přímo pomocí objektového jazyka (C++, Java, Smalltalk) využitím jeho vlastní syntaxe. Navíc každý objekt v systému automaticky obdrží identifikátor (OID), který je jednoznačný a neměnný během existence objektu. Objekt může mít buď vlastni OID, nebo může ukazovat na jiný objekt.

  Výhody:

- Operace na složitých objektech
- Rekurzivní struktury
- Abstraktní datové typy
- Rozhraní k OO jazyku
- Složité transakce

Objektově-relační databáze

"Rozšířená relační" a "objektově-relační" jsou synonyma pro databázové systémy, které se snaží sjednotit rysy jak relačních, tak objektových databází. ORDB je specifikována v rozšíření SQL standardu — SQL3. Do této kategorie patří např. Informix, IBM, Oracle a Unisys.

- Datový model

  ORDB využívají datový model tak, že "přidávají objektovost do tabulek". Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu, nazývanou abstraktní datové typy (ADT). ADT je datový typ, který vznikne zkombinováním základních datových typů. Podpora ADT je atraktivní, protože operace a funkce asociované s novými datovými typy mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu. ORDB jsou nadmnožinou RDB a pokud nevyužijeme žádné objektové rozšíření jsou ekvivalentní SQL2. Proto má omezenou podporu dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem.

- Dotazovací jazyk

  ORDB podporuje rozšířenou verzi SQL — SQL3. Důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod. ORDB je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření, pracuje právě s nimi.

- Výpočetní model

  Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s databází. Přímá podpora objektových jazyků stále chybí, což nutí programátory k překladu mezi objekty a tabulkami.

  Dva přístupy:

- univerzální paměť, kdy všechny druhy dat jsou řízeny SŘBD), jde o integraci (různými způsoby!) ⇒ univerzální servery
- univerzální přístup, kdy všechna data jsou ve svých původních (autonomních) zdrojích

  Technika: middleware

  - brány (min. dva nezávislé servery)
  - zobrazení schémat, transformace dotazů
  - objektové obálky: Persistence Software, Ontologic, HP,
  - Next, ... (problémy: výkon)
  - DB založené na Web

Shrnutí

Relační model je jednoduchý a elegantní, ale je naprosto rozdílný od objektového modelu. Relační databáze nejsou navrhovány pro ukládání objektů a naprogramování rozhraní pro ukládání objektů v databázi je velmi složité. Relační databázové systémy jsou dobré pro řízení velkého množství dat, vyhledávání dat, ale poskytují nízkou podporu pro manipulaci s nimi. Jsou založeny na dvourozměrných tabulkách a vztahy mezi daty jsou vyjadřovány porovnáváním hodnot v nich uložených. Jazyky jako SQL umožňují tabulky propojit za běhu, aby vyjádřily vztah mezi daty.

Naproti tomu **objektově orientovaný model** je založen na objektech, což jsou struktury, které kombinují daný kód a data. Objektové databázové systémy umožňují využití hostitelského objektového jazyka jako je třeba C++, Java, nebo Smalltalk přímo na objekty "v databázi"; tj. místo věčného přeskakování mezi jazykem aplikace (např. C) a dotazovacím jazykem (např. SQL) může programátor jednoduše používat objektový jazyk k vytváření a přístupu k metodám. Krátce řečeno, ODB jsou výborné pro manipulaci s daty.

Hlavní rozdíl je v přístupu ke vztahům

◦ v OO databázích jsou vztahy reprezentovány pomocí OIDs, což zlepšuje přístup k datům

◦ v relačních databázích jsou vztahy mezi n-ticemi specifikovány atributy se stejnou doménou.

Nevýhody OO

◦ Chabý výkon (ORM 15-20% slabší než samotný JDBC driver). Ve srovnání s relačními jsou optimalizátory pro OO DB velmi složité.

◦ Problémy se škálovatelností, neschopnost podporovat rozsáhlé systémy.

ANSI/ISO normy SQL – objektové vlastnosti jazyka SQL99.

Thursday, May 30, 2013 8:18 AM

ANSI/ISO normy SQL

Vývoj standardů SQL:
SQL86
SQL89
SQL92
SQL/Call Level Interface 95
SQL/Persistent Stored Module Language Interface 96
SQL/Java
SQL99
SQL/Object Language Bindings 2000
SQL/Management External Data 2000
SQL/OLAP
SQL/temporal
SQL/Schemata
SQL/XML
SQL/MM

Pracovní názvy:
SQL1 (>SQL86)
SQL2 (>SQL92),
SQL3 (>SQL99),
SQL4
[zdroj: http://www-kiv.zcu.cz/~jezek_ka/vyuka/DB2%202005/OO_ORDB/(Objektove%20relacni%20database.pdf]

SQL-86 (SQL 87)
První standard formalizovaný ANSI

SQL-92 (SQL2)

Standard je rozdělen na tři úrovně: *entry*, *intermediate* a *full*. Někdy je také uváděn mezistupeň mezi entry a intermediate jako *transitional*. Úrovně slouží k tomu, aby mohlo být u implementací standardu (jednotlivých databází) uvedeno do jaké míry splňují daný standard.

Změny možno klasifikovat jako:

- Entry
  Jen formální změny oproti SQL-86
- Transitional
  - Podpora různých druhů spojení jako NATURAL JOIN, INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN
  - Podpora nových datových typů DATE, TIME, TIMESTAMP and INTERVAL, including various datetime and interval features (excluding time zones)
- Intermediate
  - Podpora dlouhých identifikátorů (do 128 znaků)
  - Podpora kurzorů a směru získávání dat příkazem FETCH
  - Podpora definice a používání znakových sad
- Full
  - Podpora dočasných tabulek
  - Možnost výběru přesnosti u datových typů TIME a TIMESTAMP
  - Možnost testování pravdivostních hodnot pomocí TRUE, FALSE or UNKNOWN

- Vnořené tabulky ve FROM
- Podpora UNION JOIN a CROSS JOIN
- Podpora pro collations znakových sad
- Vylepšené udělování práv

SQL:1999 (SQL3) Jedná se o standard pro relačně-objektový dotazovací jazyk (na rozdíl od předchozích verzí, které byly pouze relační)

Jedná se o standard pro relačně-objektový dotazovací jazyk (na rozdíl od předchozích verzí, které byly pouze relační)

- Regulární výrazy
- Rekurzivní dotazy
- Triggery
- Procedurální rozšíření (Příkazy řízení běhu - LOOP, IF...)
- Objektové rozšíření
- nové typy STRING, BOOLEAN, REF, ARRAY, typy pro full-text, obrázky, prostorová data

The SQL:1999 standard introduced a number of object-relational database features into SQL, chiefly among them **structured user-defined types**, usually called just **structured types**. These can be defined either in plain SQL with CREATE TYPE but also in Java via SQL/JRT. SQL structured types allow single inheritance.

From <http://en.wikipedia.org/wiki/Structured_type>

Existují i novější standardy

- **SQL:2003** (Představeny XML-vázané funkce, window funkce, standardizované sekvence a sloupce s automaticky generovanými hodnotami)
- **SQL:2006** (SQL může být použito ve spojení s XML - možnost importu a skladování XML dat v SQL databázi, manipulaci s nimi a publikace dat v XML formě. Možnost využití XQuery)
- **SQL:2008** (ORDER BY mimo definici kurzoru, INSTEAD OF triggery, přidán TRUNCATE příkaz)
- Jednotlivé databázové servery ne vždy dodržují ANSI normu – obvykle pouze SQL-92 Entry
- Čím vice se při vývoji aplikace využijí rysy vyšší než SQL-92 Entry, tím je menší šance, že aplikace bude provozuschopná i na jiné databázi

většina z těchto rozšíření lze najít v jiných otázkách, proto zde nebudou více rozepsána.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Objektová rozšíření SŘBD Oracle

V současné době směřuje trend ve vývoji databázových systémů směrem k objektovým SŘBD, neboť objektové SŘBD umožňují snadněji a přesněji modelovat většinu z různých tříd aplikací. Proto se výrobci relačních SŘBD snaží své produkty rozšířit o některé základní vlastnosti objektových SŘBD, čímž vznikají tzv. *objektově-relační* SŘBD. Zmíněný tred se promítá i do nového standardu **SQL 3**, jehož součástí jsou i abstraktní datové typy, persistentní programové moduly a vhnízděné tabulky. Jedním z objektově-relačních SŘBD je i systém Oracle (od verze 8i), na kterém si nyní ukážeme některá objektová rozšíření relačního SŘBD.

Abstraktní datové typy

Abstraktní datový typ lze nadefinovat příkazem **CREATE TYPE**. Tento uživatelem definovaný ADT lze pak použít všude, kde jsou používány standardní datové typy SQL. Následující příklad ukazuje definici typu *t_adresa* reprezentujícího adresu obyvatele. Ukázána je i definice tabulky, kde jedním z atributů je objekt typu *t_adresa* :

CREATE TYPE t_adresa **AS OBJECT** (

```
  ulice     VARCHAR2(30),
  cislo     NUMBER(3),
  obec      VARCHAR2(30),
  psc       NUMBER(5)
);
/
CREATE TABLE obyvatele (
  prijmeni  VARCHAR2(30),
  jmeno     VARCHAR2(30),
  adresa    t_adresa
);
/
```

Informace o struktuře tabulky lze opět získat příkazem **DESC obyvatele**. Strukturu objektu zobrazíme rovněž příkazem **DESC t_adresa**.

Práce s objekty v SQL je, až na malé výjimky, víceméně intuitivní. Vkládání záznamů ukazuje spodní příklad. Instance objektu třídy t_adresa je vytvořena voláním implicitního konstruktoru :

```
INSERT INTO obyvatele (jmeno, prijmeni, adresa)
VALUES ( 'Jan', 'Pavlasek',
       t_adresa('Na hrazi', 15, 'Liptakov', 32100) );
```

S objekty lze snadno manipulovat jako s ostatními datovými typy, lze je i porovnávat :

```
SELECT  prijmeni, adresa FROM obyvatele;
SELECT  prijmeni, jmeno
  FROM  obyvatele
 WHERE  adresa = t_adresa('Na hrazi', 15, 'Liptakov', 32100);
```

V případě, že chceme přistupovat k jednotlivým datovým prvkům objektu, musíme použít aliasy :

```
SELECT  o.jmeno AS jmeno, o.prijmeni AS prijmeni,
       o.adresa.obec AS obec, o.adresa.psc AS psc
  FROM  obyvatele o;
```

Obdobně provedeme i aktualizaci objektu :

```
UPDATE  obyvatele o SET o.adresa.ulice = 'Pod hrazi',
                        o.adresa.cislo = 8
 WHERE  prijmeni = 'Pavlasek'
   AND  jmeno = 'Jan';
UPDATE  obyvatele SET adresa = t_adresa('Na hrazi', 15, 'Liptakov', 32100)
 WHERE  prijmeni = 'Pavlasek'
   AND  jmeno = 'Jan';
```

Objekt lze zrušit takto :

```
UPDATE  obyvatele SET adresa = NULL
 WHERE  prijmeni = 'Pavlasek'
   AND  jmeno = 'Jan';
```

Pozn. : *V tomto případě nelze již provést první příkaz **UPDATE** z předchozího příkladu, protože objekt již neexistuje. Druhý příkaz **UPDATE** naopak vytvoří instanci novou.*

Metody objektů

Samozřejmě, že objekty v SQL 3 mohou mít kromě datových prvků i metody. Následující ukázka slouží pro ilustraci definice objektu s několika metodami. Implementace metod je definována v příkazu **CREATE TYPE BODY**.

Rozhraní objektu :

```
CREATE OR REPLACE TYPE t_osoba AS OBJECT (
  jmeno     VARCHAR2(30),
  prijmeni  VARCHAR2(30),
  naroz     DATE,
  plat      NUMBER(7,2),
MEMBER FUNCTION vek RETURN INTEGER,
  MEMBER PROCEDURE zvys_plat(castka IN NUMBER),
  MEMBER PROCEDURE sniz_plat(castka IN NUMBER)
);
```

```
/
```

Implementace metod objektu :

```
CREATE OR REPLACE TYPE BODY t_osoba AS
MEMBER FUNCTION vek RETURN INTEGER IS
 BEGIN
  RETURN TO_NUMBER(SYSDATE - naroz)/365;
 END;
MEMBER PROCEDURE zvys_plat(castka IN NUMBER) IS
 BEGIN
  plat := plat + castka;
 END;
MEMBER PROCEDURE sniz_plat(castka IN NUMBER) IS
 BEGIN
  IF (plat - castka < 0) THEN
   plat := 0;
  ELSE
   plat := plat - castka;
  END IF;
 END;
END;
/
```

Řádek v tabulce může být reprezentován i objektem. Tabulku pak nadefinujeme jednoduše takto :
CREATE TABLE osoby **OF** t_osoba;
Pro zajímavost si zkuste zobrazit strukturu ADT *t_osoba* příkazem **DESC t_osoba** a také strukturu tabulky *osoby* příkazem **DESC osoby**.
Nyní vložíme do tabulky několik záznamů a zkusíme zavolat metodu *věk()* :

```
INSERT INTO osoby (jmeno, prijmeni, naroz, plat)
 VALUES ('Jarda', 'Kabrnak', TO_DATE('1.1.1980', 'DD.MM.RRRR'), 1234.56);
INSERT INTO osoby (jmeno, prijmeni, naroz, plat)
 VALUES ('Josef', 'Jiricka', TO_DATE('1.1.1970', 'DD.MM.RRRR'), 6543.21);
SELECT  o.prijmeni, o.jmeno, o.vek() AS vek
 FROM  osoby o;
```

Volání metod uvnitř PL/SQL bloku je stejné jako u ostatních objektově-orientovaných jazyků :

```
DECLARE
 clovek  t_osoba;
 c_ref   REF t_osoba;
BEGIN
 clovek := t_osoba('Jan', 'Machacek', TO_DATE('1.1.1975', 'DD.MM.RRRR'), 1122.33);
 clovek.zvys_plat(1000.0);
INSERT INTO osoby o VALUES(clovek) RETURNING REF(o) INTO c_ref;
UPDATE  osoby o SET plat = plat - 100.0
  WHERE  REF(o) = c_ref;
END;
```

Pole jako atributy

V praxi se často vyskytují případy, kdy je třeba uložit do jednoho atributu více hodnot. Typickým případem jsou alternativní čísla telefonu, na kterých je dosažitelná určitá osoba. Relační model dat nás nutí vytvořit novou entitu pro telefonní čísla, což se nám může oprávněně zdát poněkud nepřirozené. V takovýchto případech lze s výhodou použít pole proměnné délky - **VARRAY**. Pole **VARRAY** může obsahovat různý počet položek stejného datového typu (tzn. i objekty), který nesmí překročit definovanou velikost pole. Způsob použití polí je ukázán v následujícím příkladě :

```
CREATE TYPE t_tel_seznam AS VARRAY(5) OF VARCHAR2(30);
/
CREATE TYPE t_potomek AS OBJECT (
 jmeno  VARCHAR2(30),
 vek    NUMBER(3)
```

```
);
/
CREATE TYPE t_pot_seznam as VARRAY(10) OF t_potomek;
/
CREATE TABLE personal (
 jmeno    VARCHAR2(30),
 prijmeni VARCHAR2(30),
 telefony t_tel_seznam,
 deti     t_pot_seznam
);
```

Způsob vkládání a výběru záznamů se nemění :

```
INSERT INTO personal (jmeno, prijmeni, telefony, deti)
 VALUES ( 'Jan', 'Kokoska',
        t_tel_seznam('123', '124', '125'),
        t_pot_seznam(t_potomek('Jirka', 10), t_potomek('Jakub', 16)));
SELECT * FROM personal;
```

K jednotlivým záznamům pole lze přistupovat tak, že pole "přetypujeme" na tabulku (*un-nesting*). Všimněte si, že následující dotazy implicitně provádí spojení bázové a vnořené tabulky reprezentované polem :

```
SELECT  p.prijmeni, d.jmeno, d.vek
 FROM  personal p, TABLE(p.deti) d;
SELECT  p.prijmeni, COUNT(d.jmeno) AS pocet_deti
 FROM  personal p, TABLE(p.deti) d
 GROUP  BY p.prijmeni
 ORDER  BY p.prijmeni;
```

Nevýhodou polí je, že z hlediska aktualizace se s nimi pracuje jako s atomickými hodnotami. Ve výše uvedeném příkladě tedy nelze jednoduše přidat, změnit nebo odstranit telefonní číslo. Aktualizaci lze provést nejvýše následujícím způsobem :

```
UPDATE  personal SET telefony = t_tel_seznam('222', '333')
 WHERE  prijmeni = 'Kokoska';
```

V PL/SQL je práce s polem **VARRAY** velmi snadná. K jednotlivým položkám lze přistupovat přes index. Kromě toho každé pole obsahuje atribut **count**, jehož hodnota udává počet prvků pole :

```
DECLARE
 i  INTEGER;
 p  personal%ROWTYPE;
BEGIN
 SELECT  * INTO p
  FROM  personal
 WHERE  prijmeni = 'Kokoska';
dbms_output.new_line;
  dbms_output.put_line('Pocet telefonu : '||p.telefony.count);
dbms_output.put_line('Deti :');
  FOR i IN 1..p.deti.count LOOP
   dbms_output.put_line(p.deti(i).jmeno);
  END LOOP;
END;
```

Vhnízděné (vnořené) tabulky

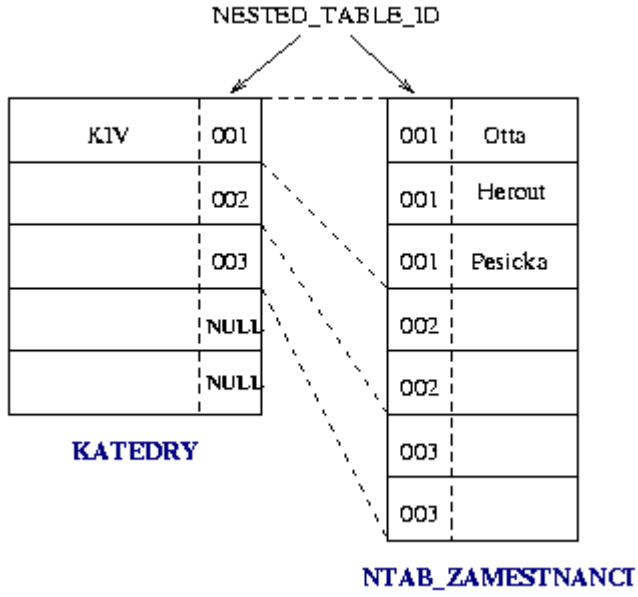
Vnořené tabulky mají oproti polím tu výhodu, že z hlediska aktualizace lze přistupovat k jednotlivým řádkům, jejichž počet není nijak omezen (teoreticky). Nevýhodou však je, že prvky (řádky) v tabulce nemají definované pořadí, jak tomu je u polí. Vnořenou tabulku vytvoříme tak, že nadefinujeme uživatelský datový typ podobně jako u polí :

```
CREATE TYPE t_zamestnanec AS OBJECT (
 jmeno    VARCHAR2(30),
 prijmeni VARCHAR2(30),
 plat     NUMBER(5)
);
```

/
CREATE TYPE tab_zamestnanci **AS TABLE OF** t_zamestnanec;
/
Definujeme-li tabulku, která obsahuje vnořenou tabulku, musíme definovat jméno tabulky, kde budou fyzicky uloženy záznamy z vnořených tabulek. V uvedeném příkladě to je tabulka NTAB_ZAMESTNANCI :

```
CREATE TABLE katedry (
 cislo_kat   NUMBER(5),
 nazev       VARCHAR2(50),
 zamestnanci tab_zamestnanci
) NESTED TABLE zamestnanci STORE AS ntab_zamestnanci;
```

Ve skutečnosti jsou vnořené tabulky řešeny "klasickým způsobem" - vazba mezi mateřskou a vnořenou tabulkou je realizována přes skrytý sloupec NESTED_TABLE_ID, jak ukazuje spodní obrázek :



Přestože tabulka NTAB_ZAMESTNANCI je viditelná v katalogu (příkazem **SELECT * FROM** tab;), není přístupná pro příkazy DML. Zrušíme-li mateřskou tabulku KATEDRY, je automaticky zrušena i tabulka NTAB_ZAMESTNANCI. Vkládání záznamů je stejné jako u polí :

```
INSERT INTO katedry (cislo_kat, nazev, zamestnanci)
  VALUES (111, 'KIV',
        tab_zamestnanci(
          t_zamestnanec('Max', 'Otta', 12345),
          t_zamestnanec('Pavel', 'Herout', 12345),
          t_zamestnanec('Ladislav', 'Pesicka', 12345)));
```

Dotazy nad vnořenými tabulkami :

```
SELECT * FROM katedry;
SELECT  z.*
  FROM  katedry k, TABLE(k.zamestnanci) z
 WHERE  k.cislo_kat = 111;
```

Aktualizace vnořených tabulek :

```
INSERT INTO TABLE( SELECT  zamestnanci
                   FROM  katedry
                  WHERE  cislo_kat = 111 )
  VALUES ('Martin', 'Simek', 12345);
---
UPDATE TABLE( SELECT  zamestnanci
              FROM  katedry
             WHERE  cislo_kat = 111 )
  SET plat = 9999
WHERE jmeno = 'Max';
```

```
---
UPDATE TABLE( SELECT  zamestnanci
              FROM  katedry
             WHERE  cislo_kat = 111 ) z
  SET VALUE(z) = t_zamestnanec('Martin', 'Simacek', 1234)
WHERE z.prijmeni = 'Simek';
---
DELETE FROM TABLE( SELECT  zamestnanci
                   FROM  katedry
                  WHERE  cislo_kat = 111 )
  WHERE jmeno = 'Max';
```

Vnořenou tabulku zrušíme takto :

```
UPDATE  katedry
   SET  zamestnanci = NULL
 WHERE  cislo_kat = 111;
```

Opětovné vytvoření prázdné, nebo naplněné vnořené tabulky :

```
UPDATE  katedry
   SET  zamestnanci = tab_zamestnanci()
 WHERE  cislo_kat = 111;
UPDATE  katedry
   SET  zamestnanci = tab_zamestnanci(t_zamestnanec('Max', 'Otta', 9999))
 WHERE  cislo_kat = 111;
```

Reference na objekty

V některých případech nechceme v atributu uchovávat celý objekt, ale pouze referenci (ukazatel, odkaz) na něj. Proto byl v SQL3 zaveden typ *reference*. Uvažme případ, že máme danou databázi pracovišť a předmětů (nábytek apod.) uložených na jednotlivých pracovištích. Vzhledem k centrální správě předmětů by bylo poněkud nevýhodné mít u každého pracoviště vnořenou tabulku předmětů. Navíc předměty mohou být rozděleny do několika kategorií a tedy i tabulek. Následující příklad ukazuje nastíněnou situaci, kdy předměty jsou ukládány v jediné tabulce a u každého předmětu je reference na pracoviště, na němž se nachází :

```
CREATE TYPE t_pracoviste AS OBJECT (
  zkratka  VARCHAR2(3),
  nazev    VARCHAR2(50)
);
/
CREATE TABLE kancelare OF t_pracoviste;
CREATE TYPE t_predmet AS OBJECT (
  nazev      VARCHAR2(30),
  ev_cislo   NUMBER(5),
  pracoviste  REF t_pracoviste
);
/
CREATE TABLE predmety OF t_predmet;
```

Několik poznámek k referencím : každý objekt v SŘBD je jednoznačně identifikován pomocí **OID** - *Object IDentifier*. OID je jednoznačný v rámci celého SŘBD, v distribuovaných SŘBD v rámci celé sítě. Je-li určitý objekt zrušen, jeho OID již není nikdy použito pro jiný objekt. Typicky je OID generován na základě jednoznačného identifikátoru hostitelského uzlu (např. HW adresa síťového adaptéru uzlu), časové značky (aktuální čas na uzlu) a sekvenčního čísla.

Pro ilustraci si vyzkoušejte vytvořit nový objekt a vypsat jeho OID (referenci) :

```
INSERT INTO kancelare (zkratka, nazev)
  VALUES ('KIV', 'Kancelar KIV');
SELECT REF(k)
  FROM kancelare k
 WHERE zkratka = 'KIV';
```

Reference na objekty nelze považovat za referenční integritní omezení. Referenční IO omezují rozsah hodnot atributu (většinou cizího klíče), kdežto reference je chápána jako ukazatel na určitý objekt, který ovšem nemusí existovat (vzpomeňte si na ukazatele v C ;-). Nicméně u referencí lze

specifikovat rozsah, a to ve smyslu množiny objektů ve specifikované tabulce :

```
DROP TABLE predmety;
CREATE TABLE predmety OF t_predmet (SCOPE FOR (pracoviste) IS kancelare);
```

Referenci na určitý objekt vložíme do záznamu takto :

```
INSERT INTO predmety (nazev, ev_cislo, pracoviste)
  VALUES ('Zidle', 555,
        (SELECT REF(k) FROM kancelare k WHERE k.zkratka = 'KIV'));
```

V případě, že se budeme snažit vložit do sloupce PRACOVISTE v tabulce PREDMETY referenci na objekt z jiné tabulky než KANCELARE, databáze vkládání záznamu odmítne hláškou :

ORA-22889: REF value does not point to scoped table

Přístup k objektům přes reference (dereference) je velmi snadný :

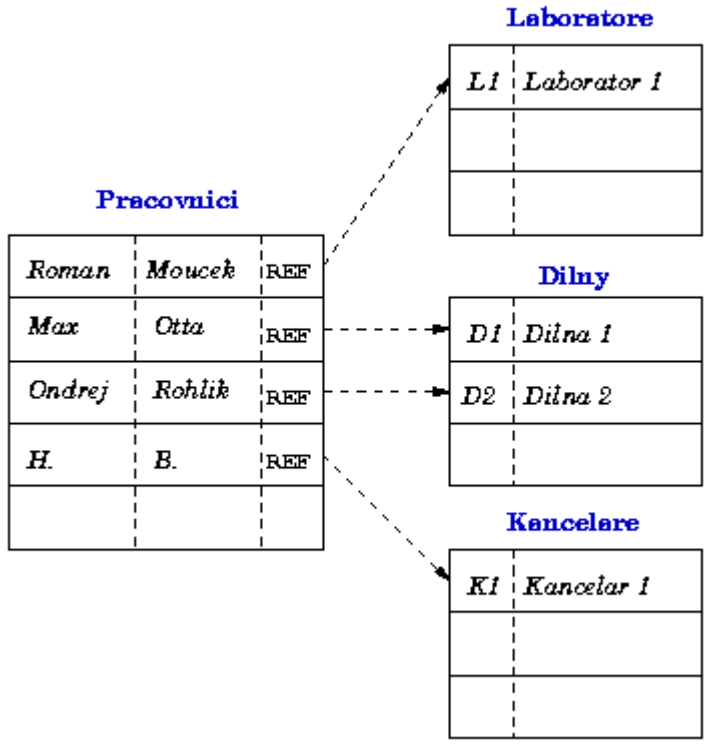
```
SELECT  p.nazev, p.ev_cislo, p.pracoviste.zkratka
  FROM  predmety p;
```

Reference na neexistující objekty lze najít také snadno :

```
DELETE FROM kancelare;
SELECT  p.nazev
  FROM  predmety p
 WHERE  p.pracoviste IS DANGLING;
```

Příklad : evidence zaměstnanců a jejich pracovišť

Uvažme databázi nějakého ústavu, kde jsou centrálně evidování zaměstnanci, jednotlivá pracoviště jsou evidována zvlášť. Vazba mezi pracovníkem a jeho pracovištěm je realizována referencí u záznamu pracovníka na záznam jeho pracoviště. Celou situaci ilustruje obrázek :



Z předchozích příkladů využijeme typ **t_pracoviste** a tabulku **KANCELARE**. Zbývající tabulky vytvoříme takto :

```
CREATE TABLE dilny OF t_pracoviste (
  PRIMARY KEY (zkratka)
);
CREATE TABLE laboratore OF t_pracoviste (
  PRIMARY KEY (zkratka)
);
CREATE TYPE t_pracovnik AS OBJECT (
  prijmeni    VARCHAR2(30),
  jmeno       VARCHAR2(30),
  pracoviste  REF t_pracoviste
```

```
);
/
CREATE TABLE pracovnici OF t_pracovnik;
```
Tabulky ještě naplníme daty :
```
INSERT INTO dilny VALUES ('D1', 'Dilna 1');
INSERT INTO dilny VALUES ('D2', 'Dilna 2');
INSERT INTO kancelare VALUES ('K1', 'Kancelar 1');
INSERT INTO laboratore VALUES ('L1', 'Laborator 1');
INSERT INTO pracovnici
  VALUES ('Max', 'Otta',
        (SELECT REF(p) FROM dilny p WHERE zkratka = 'D1'));
INSERT INTO pracovnici
  VALUES ('Ladislav', 'Pesicka',
        (SELECT REF(p) FROM dilny p WHERE zkratka = 'D1'));
INSERT INTO pracovnici
  VALUES ('Ondrej', 'Rohlik',
        (SELECT REF(p) FROM dilny p WHERE zkratka = 'D2'));
INSERT INTO pracovnici
  VALUES ('Helena', 'Benesova',
        (SELECT REF(p) FROM kancelare p WHERE zkratka = 'K1'));
INSERT INTO pracovnici
  VALUES ('Roman', 'Moucek',
        (SELECT REF(p) FROM laboratore p WHERE zkratka = 'L1'));
COMMIT;
```
Seznam všech zaměstnanců včetně jejich příslušnosti k pracovišti na závěr vypíšeme jistě elegantním způsobem :
```
SELECT  p.prijmeni, p.jmeno, p.pracoviste.nazev
  FROM  pracovnici p;
```

From <http://www.kiv.zcu.cz/~zima/vyuka/db2/sql99.html>

Objektově relační databáze.

Thursday, May 30, 2013 8:19 AM

http://www.fi.muni.cz/~xbatko/oracle/compare.html

Za posledních 25 let je mohutný trend přechodu od strukturovaného k objektově orientovanému programování, a to i v oblasti zpracování dat a databází. Objektově orientované DB se objevili v 90. Letech minulého století a kladou si za cíl urychlit a ulehčit práci s daty. Situace ovšem není jednoduchá, protože relační a objektový přístup je od základu rozdílný. Existuje tak mnoho výhod i mnoho nevýhod pro relační i objektové DB. Na současném trhu existují 3 základní typy:

- Relační DB (Relational Database Management System, RDBMS) – výkonné na tradičních datech. Př. Oracle 7.x, DB2.
- Objektově-relační (Object Relational Database Management t System, ORDBMS) – Poskytuje programátorské pohodlí, rychlý a udržitelný vývoj aplikací. Př. Oracle 8.x, 9.x, 10.x.
- Objektové DB (Object Database Management system, ODBMS) – Výkonné na netradičních datech, slabší v databázových rysech. Př. Jasmine, Gemstone, O2.

Objektově relační databáze

Objektově relační DB se snaží přinést objektové rysy do relačních databází. ORDBMS je specifikována v rozšíření SQL standardu – SQL3. Do této kategorie patří např. Informix, IBM, Oracle a Unisys.

Objektově relační mapování přináší vrstvy mezi OO aplikací a SQL databází. Charakteristiky jsou:

Myšlení v objektech, ev. Cache objektů, zodpovědné za persistenci.

Datový model ORDBMS

Rozšiřují datový model tak, že přidávají objektovost do tabulek. Všechny trvalé informace jsou stále v tabulkách, ale některé položky mohou mít bohatší datovou strukturu (porušení 1. NF), nazývanou abstraktní datové typy, tzv. ADT:

- ADT je datový typ, který vznikne kombinací základních datových typů
- Podpora dědičnosti, polymorfismu, referencí a integrace s programovacím jazykem je omezená
- Funkce a operace jsou asociované s novými datovými typy, mohou být použity k indexování, ukládání a získávání záznamů na základě obsahu nového datového typu.

Dotazovací jazyk

ORDBMS podporují rozšířenou verzi SQL – SQL3 (SQL 99), důvodem je podpora objektů (tj. dotazy obsahující atributy objektů). ORDBMS je stále relační, protože data jsou uložena v řádcích a sloupcích tabulek a SQL, včetně zmíněných rozšíření. Typická rozšíření zahrnují dotazy obsahující vnořené objekty, atributy, abstraktní datové typy a použití metod.

Jazyk SQL s rozšířením pro přístup k ADT je stále hlavním rozhraním pro práci s DB.

Objektové vlastnosti SQL99 (SQL3)

Objektové rozšíření standardizované v SQL99 zahrnuje:

- Strukturované uživatelské typy – Oracle od verze 9.i včetně jednoduché dědičnosti. Mohou být organizovány do hierarchie s děděním. Chování uživatelem definovaných typů je realizováno pomocí procedur a funkcí a (metod u ADT). Jedná se o řádkové typy, ADT, odlišující typy.
- Pole s proměnnou délkou (VARRAY) – CREATE TYPE typ AS VARRAY(5) OF VARCHAR(15)
- Hnízděné tabulky – typ TABLE
- Typ REF – odkaz ukazatel – jeho obsahem je OID nějakého záznamu, nelze s ním manipulovat jako s hodnotou, ale jako s odkazem. Zajišťuje objektovou identitu.
    - Výhoda - pro sdílení objektů (nejsou zbytečně kopírována data a změna se provádí na jednom místě)

- **CREATE TYPE TypHerec AS (jmeno CHAR(30), nejlepsiFilm REF (FilmTyp))**
- Dereference př. SELECT Film->titul FROM hrajev WHERE herec->jmeno='Chaplin';

SQL99 je kompatibilní s existujícími jazyky a další vlastnosti jsou:

- Řádkové typy (jsou typem relace)
  - Vytvoření řádkových typů:

```
CREATE ROW TYPE typadresa (ulice CHAR VARYING(50), mesto CHAR
VARYING(20));
```

  - Příklad tvorby tabulky s řádkovým typem:

```
CREATE TABLE FilmovyHerec OF typherec;
CREATE TABLE FilmovyHerec (
jmeno CHAR VARYING(30),
adresa ROW (
ulice CHAR VARYING (50),
mesto CHAR VARYING(20)
)
);
```

- Tvorba dotazů:

```
SELECT FilmovyHerec.jmeno, FilmovyHerec.adresa.ulice
FROM FilmovyHerec WHERE FilmovyHerec.adresa.mesto = 'Plzeň';
```

- Lze definovat i podtypy a podtabulky CREATE TYPE typSekretarka UNDER typZamestnanec AS(...
- Abstraktní datové typy (jsou typem atributu relace) – umožňují zapouzdření atributů a operací (na rozdíl od řádkových typů). Hodnoty jejich typů mohou být umístěny do sloupců tabulek.
  - Př. (Oracle)

```
CREATE TYPE typZamestnanec AS (
c_zam INTEGER,
METHOD mzda() RETURNS DECIMAL);
CREATE METHOD mzda ... FOR typZamestnanec
BEGIN ... END
```

  - Instance vznikají konstruktorem jmenoTypu(), operátorem NEW jmeno hodnota, příkazem INSERT INTO osoby VALUES(...)
  - Funkce a procedury vyjádřeny v SQL/PSM (Persistent stored module), nebo C/C++, Java, ADA.... Jsou svázány s ADT. Metody jsou uložené ve schématu typu definovaného uživatelem. Metody se dědí. Metody i funkce mohou být polymorfní (liší se způsobem výběru). CREATE PROCEDURE zjisti_cenu ...; CALL zjisti_cenu(...);
- Odlišující typy (musí být FINAL) – emulace domén – strong typing
- OID – záznamy mají/mohou mít OID (v relačních DB mohou být použity jako primární klíče), které zajišťují objektovou identitu. V jiných záznamech atribut typu REF – odkaz ukazatel. Zpřístupnění klauzulemi REF IS SYSTEM GENERATED a REF IS USER GENERATED.

Srovnání DBMS

Srovnání databázových systémů			
Kritérium	**RDBMS**	**ORDBMS**	**ODBMS**
Definovaný standard	SQL2 (ANSI X3H2)	SQL3/4 (in process)	ODMG-V2.0
Podpora pro objektově orientované programování	Špatná; programátoři stráví 25% času kódování mapováním objektového programu do databáze	Omezená hlavně na nové datové typy	Přímá a rozsáhlá
Jednoduchost používání	Strukturám tabulky je jednoduché porozumět;	Totéž co RDBMS, navíc s nějakými matoucími	OK pro programátory; nějaký SQL přístup pro

	mnoho dostupných nástrojů pro koncové uživatele	rozšířeními	koncové uživatele
Jednoduchost vývoje	Poskytuje nezávislost dat z aplikace, dobrá pro jednoduché vztahy	Poskytuje nezávislost dat z aplikace, dobrá pro jednoduché vztahy	Objekty jsou přirozenou cestou k modelu; může vyhovět širokým rozsahem typů a vztahů
Rozšiřitelnost a obsah	Žádná	Omezená hlavně na nové datové typy	Může pracovat s libovolnou složitostí; uživatelé mohou psát metody a jakékoliv struktury
Složité datové vztahy	Pro model obtížné	Pro model obtížné	Může pracovat s libovolnou složitostí; uživatelé mohou psát metody a jakékoliv struktury
Výkon versus spolupracovatelnost	Úroveň bezpečnosti se mění s dodavatelem, je třeba vzájemně porovnat; dosáhnutí obojího vyžaduje rozsáhlé testování	Úroveň bezpečnosti se mění s dodavatelem, je třeba vzájemně porovnat; dosáhnutí obojího vyžaduje rozsáhlé testování	Úroveň bezpečnosti se mění s dodavatelem; většina ODBMSs dovoluje programátorům rozšířit funkčnost DBMS definováním nových tříd
Distribuce, replikace, a spojené databáze	Rozsáhlá	Rozsáhlá	Podle dodavatele; pár jich poskytuje rozsáhlou podporu
Vyspělost produktu	Velmi vyspělé	Nezralé; rozšíření jsou nová, stále se definují a jsou relativně neprozkoušená	Relativně vyspělé
Podpora pro lidi a univerzálnost SQL	Široká podpora nástrojů a trénovaných vývojářů	Může využívat výhod nástrojů RDBMS a vývojářů	Vybaveno SQL, ale určeno pro objektově orientované programování.
Softwarové ekosystémy	Poskytováno hlavními RDBMS společnostmi	Poskytováno hlavními RDBMS společnostmi	ODBMS výrobci začínají emulovat RDBMS výrobce, ale žádný nenabízí velký obchod jiným ISV
Životaschopnost výrobce	Očekávaná pro hlavní zaběhnuté RDBMS výrobce	Očekávaná pro hlavní RDBMS výrobce; UniSQL bojuje	Menší než se čekalo; stále se očekává zmenšování
Zdroj: International Data Corporation, 1997			

From <http://www.fi.muni.cz/~xbatko/oracle/compare.html>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Vlastnosti objektově orientovaného datového modelu.

Thursday, May 30, 2013 8:19 AM

The Object-Oriented Data Model

1. A data model is a logic organization of the real world objects (entities), constraints on them, and the relationships among objects. A DB language is a concrete syntax for a data model. A DB system implements a data model.
2. A core object-oriented data model consists of the following basic object-oriented concepts:
   (1) **object and object identifier**: Any real world entity is uniformly modeled as an object (associated with a unique id: used to pinpoint an object to retrieve).
   (2) **attributes and methods**: every object has a state (the set of values for the attributes of the object) and a behavior (the set of methods - program code - which operate on the state of the object). The state and behavior encapsulated in an object are accessed or invoked from outside the object only through explicit message passing.
   [An attribute is an instance variable, whose domain may be any class: user-defined or primitive. A class composition hierarchy (aggregation relationship) is orthogonal to the concept of a class hierarchy. The link in a class composition hierarchy may form cycles.]
   (3) **class**: a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class (instance-of relationship). A class is similar to an abstract data type. A class may also be primitive (no attributes), e.g., integer, string, Boolean.
   (4) **Class hierarchy and inheritance**: derive a new class (subclass) from an existing class (superclass). The subclass inherits all the attributes and methods of the existing class and may have additional attributes and methods. single inheritance (class hierarchy) vs. multiple inheritance (class lattice).

From <http://www.cs.sfu.ca/CourseCentral/354/zaiane/material/notes/Chapter8/node3.html>

HDM, SDM a RDM = záznamově orientované modely. V 90. letech se začaly objevovat první objektově orientované SŘBD (OOSŘBD), které umožňují pracovat s datovou abstrakcí na úrovni objektů.

- Výhody
  - snadnější aktualizace dat
  - přímé vyjádření složitých objektů modelované reality v databázi (odpadají mezikroky převodu objektů do normalizovaných tabulek relační databáze. Stejně tak je zjednodušen i opačný krok načítání objektů z databáze do aplikace)
  - součástí uložených objektů je také jejich chování (metody atd.)
- Nevýhoda
  - Vývoj a návrh objektově orientovaných modelů v jejich univerzálnosti a komplexnosti je velmi složitý proces ⇒ menší uplatnění tohoto typu modelu v reálných aplikacích.

Objektově orientované programování (OOP)

Koncepce objektově orientovaných databází vychází z principů používaných v OOP - základem je objekt (prvek typu třída). Data se nazývají atributy, funkce metody (služby). Metoda je aktivována příchodem zprávy do jiného objektu.

Hlavní vlastnosti OOP: zapouzdření dat, dědičnost, polymorfismus

Objektově orientované modelování dat

Postup objektově orientovaného modelování dat lze stručně shrnout do následujících bodů:

- Vyhledají se objekty jako nositelé aktivit (např. metodou gramatické inspekce: podstatná jména představují objekty nebo třídy, přídavná jména představují hodnoty atributů a slovesa představují většinou aktivity).
- Identifikují se třídy zobecňování objektů se stejnými atributy (+ zkoumá se možnost uspořádat třídy hierarchicky podle dědičnosti) a vztahy.
- Stanoví se integritní omezení na hodnoty jednotlivých atributů a případně na typy atributů.
- Vyhotoví se seznam nabízených a požadovaných služeb pro všechny metody (přehled toku zpráv)
- Implementují se metody (teprve po jednoznačném vymezení všech funkcí systému)

Hlavní rozdíly mezi RDM a ODM:

	RDM	ODM
1	▪ relační tabulka ▪ jeden záznam ▪ manipulace s atributy záznamu	▪ množina objektů ▪ jeden objekt ▪ přenos a zpracování zpráv
2	normalizace relací (dekompozice) vede k rozptýlení popisu vlastností složitého objektu do mnoha tabulek	spojuje jednotlivé složky pomocí odkazů
3	záznamy relací jsou omezeny na jednoduché datové typy	složité strukturované datové entity - objekty, které lépe vystihují prvky reálného světa
4	manipulace s hodnotami atributů záznamů	operace posílání zpráv poskytuje větší možnosti
5	každá tabulka musí mít identifikační klíč (ten nemusí odrážet požadavky zadání)	zabezpečuje identifikaci objektů vlastními systémovými prostředky (OID)
6	při zpracování dotazů dochází často k získávání údajů z několika tabulek ⇒ narůstá čas potřebný k vyhodnocení dotazu	ke spojování množin dochází v daleko menší míře; dotazovací konstrukce lze díky polymorfismu aplikovat i na množiny obsahující různé typy objektů

Pozn.: RDM za určitých podmínek představuje zvláštní případ ODM.

Produkty: Objectivity, Versant, POET, CACHÉ, db4o, Ozone, GOODS, XL2, ZODB, Prevayler

Bariéry rozšíření objektových databázových systémů:

- neochota vývojářů a jejich klientů k přechodu od tradičního relačního přístupu k objektovému
- nedostatek kvalifikovaných vývojářů
- nízká podpora standardů
- nízká podpora dotazovacích jazyků
- neexistence mechanismu pro řízení přístupu k datům

Původní text (dle mého "mimo mísu")

Objektový (konceptuální model)
- představuje statický model reality (businessu)
- popisuje z čeho je realita složena a jaké jsou základní (podstatné/statické) složky (objekty) a vazby mezi nimi.
- Akce (metody) a algoritmy, vázané k objektům v jejich životních cyklech, zde mají význam též statický (jsou podřízeny statickému - pohledu).
- K popisu lze využít specifický diagram – Diagram tříd (Class Diagram - základní diagram jazyka UML).
- Model (entitních, business, analytických) objektů (podstata struktury reality) sleduje základní stavební kameny, z nichž se realita (problémová doména) skládá.

Diagram tříd
- Původní strukturální přístup k analýze IS spočíval v rozdělení systému na funkční a datovou část (např. DFD - Data Flow diagramy a ER - Entity Relationship model).

- Přínosem byla funkční hierarchická dekompozice – psaní programu shora dolů a datové konceptuální modelování.
- Rostoucí složitost systémů (magická hranice 1000 entit a 10000 funkcí) znemožňuje soudržnost datové a funkční vrstvy.
- Objektový přístup čelí složitosti systému tím, že třída (objekt) jako nositel (funkční) odpovědnosti (dovednosti), plně odpovídá za svá data.
- Objekt má svou identitu, vlastnosti, chování a odpovědnost. Síla odpovědnosti spočívá v tom, že je nedělitelná – žádný jiný objekt nemůže odpovědnost sdílet – dělit se o ni, plést se do ní.
- Modelování tříd a objektů je klíčová aktivita objektově orientovaného vývoje.

Třída, Objekt

- **Třída** - popis množiny objektů sdílejících stejné vlastnosti (atributy), chování (operace/metody) a vztahy.
- **Objekt** - instance třídy (chybně se pojem třída a objekt volně zaměňují).

Definice – J. Rumbauh: objekt je diskrétní entita s jasně definovaným rozhraním, které zapouzdřuje stav a chování.

- Třídu si můžeme představit jako razítko, objekty jsou pak otisky tohoto razítka, které vidíme na papíře.
- Při návrhu třídy neuvažujeme o konkrétním naplnění atributů, pouze určíme jejich název a typ. Teprve při vzniku instance objektu se atributům přiřadí skutečné hodnoty.
- Třída je jednoznačně určena svým názvem (v příslušném názvovém prostoru – balíčku). Pro třídu je možno definovat vlastnosti - atributy (Attribute) a chování - operace (Operation).
- Hledání tříd, jejich atributů a kompetencí - vyberme z reality objekty, kandidáty pro zobecnění na třídy a prověřme jejich vhodnosti:
  - Potenciální třída je smysluplná, pokud je nezbytná pro funkci systému.
  - Potenciální třída je dostatečně stabilní a invariantní vůči vnějším změnám např. technologie, legislativy apod.

Hledání tříd na základě analýzy podstatných jmen a sloves.

Analyzujeme jazyk problémové domény, např. text sebraných požadavků. Podstatná jména a jejich spojení mohou označovat třídy nebo atributy. Slovesa mohou označovat odpovědnosti, chování tříd.

Pozor na skryté, utajené třídy, které nejsou v textu uvedeny.

Klasifikace, zařazení do třídy přenáší význam na formální objekt, je nositelem sémantiky. Klasifikace je jedním z nejdůležitějších způsobů, jímž lidé uspořádávají, vnímají, chápou okolní svět. Existuje mnoho způsobů jak klasifikovat okolní svět, proto je analýza tak náročná.

Atributy

Definice atributů

Atribut určuje vlastnosti objektu, je nositelem informace o objektu.

Atributy popisují hodnoty (stavy) udržované v jednotlivých objektech.

Objekty jsou vymezeny (popsány) množinou atributů.

Atributy popisují vlastnosti objektů, které potřebujeme k dosažení daného cíle. Reálný objekt ve své nekonečné složitosti nelze vymezit omezenou množinou atributů. Základní problém analýzy IS - výběr rozumného množství relevantních atributů.

S atributy mohou manipulovat výhradně služby daného objektu.

Klíčovou otázkou je zodpovědnost určitého objektu za uchování informací. Hledání atributů je řízeno otázkami: Jak je objekt popsán v kontextu zodpovědností daného systému. V jakých stavech se může objekt v průběhu svého životního cyklu nacházet

Specifikace atributů

Atribut je definován: jménem, typem (formátem) viditelností (veřejný – public, soukromý – private a

chráněný – protected).

Každý atribut pečlivě pojmenujte. Volte názvy, které jsou běžné v aplikační oblasti a jsou rozumné délky a pevné struktury. Ke každému atributu připojte vysvětlující text.

Hledejte omezující podmínky pro hodnoty atributů. Omezení se vztahují na: formáty, rozsah, výčet přípustných hodnot, přesnost implicitní hodnoty, požadavek na nastavení výchozích hodnoty atributu prevalidační a postvalidační podmínky – podmínky, které musí být splněny před a po změně hodnoty atributu, za jakých podmínek je povolen přístup k atributu (např. v závislosti na hodnotách ostatních atributů) závislost atributů, viz datové modelování, jak změna jednoho atributu ovlivňuje hodnoty jiných – závislých atributů, viz normalizace relačního modelu.

Identita objektu

Objekt je vedle svého stavu a chování jednoznačně určen, je jedinečný, má identitu, atribut, který jej jednoznačně identifikuje mezi všemi ostatními objekty dané třídy, má své ID. Atribut zajišťující identitu, se v datovém modelování nazývá primární klíč.

Čtenář (číslo čtenáře, jméno, adresa, kontakt) Kniha (isbn, autor, titul) Exemplář (číslo exemp, datum nákupu) Exempláře knihy se liší inventárním číslem a sledujeme u nich datum nákupu.

Problematika volby primárního klíče,

Umístění atributů

Ve třídách, které jsou vázány dědičností (generalizace) umístíme atribut do co nejvyšší třídy, ve které atribut platí pro všechny její generické podtřídy (specializace).

Hledání tříd, doporučení

Každá třída by měla mít 3-5 klíčových odpovědností První extrém - není dobré, když existuje velké množství malých tříd Druhý extrém – není dobré mít velké třídy Nezavádějte „funktiody“ – pro jednotlivé funkce systému nezavádějte třídy Vyhýbejte se stromům dědičnosti s mnoha úrovněmi

<u>Vazby – relace mezi třídami</u>

Vazba asociace (Association)

Vazba asociace mezi třídami je vyjádřením abstraktního vztahu mezi objekty (instancemi tříd). Asociace říká, že objekty mají mezi sebou přímý vztah, že o sobě ví.

Zaměstnanec pracuje v daném oddělení, mohu se ptát: V jakém oddělení pracuje zaměstnanec, mohu získat seznam všech zaměstnanců v oddělení. Vazba je nositelem významu – sémantiky, odpovídá – mapuje požadavky kladené na systém. Je trvalejšího charakteru.

Asociace je společný typ vazby pro:

- agregaci, vyjadřující vztah mezi celkem a částí
- prostou asociaci, vyjadřující prostou objektovou referenci.

Vazba asociace je specifikována řadou vlastností, z nichž některé jsou vázány přímo k vazbě asociace (například název), ostatní k zakončením vazby (například role). Podrobné určení vlastností až v okamžiku návrhu – specifikace návrhových tříd a jejich vazeb má „implementační“ důsledky.

Vazbu asociace lze zavézt jako orientovanou (Navigability), přičemž neorientovaná vazba je považována za obousměrnou (dva jednosměrné vztahy).

Třídy v asociaci mohou vůči sobě vystupovat v rolích (Role) (Objednávka – Zaměstnanec, Zaměstnanec vystupuje ve vztahu k objednávce v roli Prodejce). Každá strana asociace má své jméno – roli. Role popisuje vlastnost, funkci třídy „viděné“ z druhé strany.

V asociaci lze určit násobnost vazby, (kardinalitu) multiplicitu, která vyjadřuje počet možných vazeb objektů tříd v asociaci (0, 0..1, 0..*, 1, 1..*, *, M..N, …).

Násobnost vazby definuje, kolik může k jednomu objektu, tj. k jedné instanci třídy A na jedné straně vztahu existovat minimálně (parcialita) a maximálně (kardinalita) objektů ze třídy B na druhé straně vztahu a obráceně.

- Standardně je vazba asociace implementována zavedením atributu třídy - role pro zachycení objektové reference (množiny referencí pro parcialitu 0..*) na objekty druhé třídy. Implementace vazby – objekt si sebou nese reference na asociované objekty.
- S vazbami je třeba šetřit.
- Na rozdíl implementace v relačním datovém modelu se jedná o explicitní vyjádření vazby. V relačním modelu se pro vyjádření vazby používají tzv. cizí klíče – implicitní implementace vazby.

Další vlastnosti vazeb související s implementací vazby (mimo UML, CASE):

Každý konec asociace, vazby se nazývá role. Pro roli můžeme definovat řadu vlastností.

- Asociativní třída (Association Class) je vazba asociace, která je rozšířena přiřazením třídy pro zachycení informací nutných pro úplnou specifikaci této vazby.
- Asociativní třída se používá například v případě oboustranně násobné vazby N:M. Asociativní třída nemá vlastní identitu, identitu přejímá od „asociovaných“ tříd
- Vztah mezi třemi a více prvky popisují vícenásobné asociace (N-ary Association). Pro vyjádření vícenásobné asociace se používá element modelu asociativní třída.

Vazba agregace (Aggregation)

Agregace je vyjádřením abstrakce vztahu mezi objekty (instancemi tříd), který odpovídá vztahu celku a části .

Agregace je speciálním případem asociace. Jazyk UML rozlišuje mezi dvěma typy agregací:

- prostou agregací (Simple Aggregation)
- kompozicí (Composition).

Vazba kompozice je silnější než vazba prosté agregace, jedná o vlastnictví částí celkem. Pokud je celek existenčně (tj. svou logikou) závislý na částech a nemůže bez nich fungovat, jedná se o kompozici – pokud se bez nich obejde, jedná se o agregaci.

Agregace: Profesoři - Katedra - zruším katedru, profesoři zůstanou, mohou fungovat bez katedry

Kompozice: Fakulta - katedra, zruším fakultu, katedra je bezvýznamná

Vazba generalizace (Generalization)

Vazba generalizace je vyjádřením vztahu mezi obecným elementem (Parent) a specifickým elementem (Child), který je konzistentní s obecným elementem a přidává k jeho definici další informace, je tedy bližší specifikací (specializací) obecného elementu.

Vazba generalizace mezi třídami je vyjádřením vlastnosti dědičnosti, jedné ze základních vlastností objektově orientovaného přístupu.

Jazyk UML povoluje vyjádřit vícenásobnou dědičnost zavedením více vazeb generalizace.

Vazba závislosti (Dependency)

- umožňuje znázornit jistou závislost mezi elementy modelu.
- je určena svým názvem a obvykle se používá s určitým stereotypem, který blíže specifikuje formu závislosti, zavádí její typ.
- je znázorněna orientovanou přerušovanou čárou, kde orientace je vyjádřena šipkou ve směru závislosti.

Závislost obvykle vzniká pouze dočasně pro potřeby poskytnutí služby klientskému objektu a poté

tato vazba zaniká (implementační rozdíl od asociace).

Změna jednoho (nezávislého) elementu ovlivní druhý (závislý) element.

Chování objektů, předávání zpráv

Objekt poskytuje služby prostřednictvím operací (metod).

Rozhraní objektu je množina operací, které nabízí objekt k použití pro jiné objekty (nebo externí agenty). Objekty jsou známy jiným objektům pouze prostřednictvím svého rozhraní. Objekt má i své vnitřní – interní operace, které slouží k udržení vnitřní konzistence (stavu) objektu.

Objekt může poskytovat více rozhraní – mít více rolí, podle kontextu ve kterém se nachází. Stejně jako v reálném světě člověk vystupuje v různých rolích podle toho, v jakém kontextu se právě nachází (v zaměstnání se nachází v roli pracovníka, doma manželem, v automobilu řidičem)

Objekty tak odbourávají nevýhodu strukturálních metod, spočívající ve vzájemné izolaci funkční a datové vrstvy.

Objekty spolupracují proto, aby společně mohly vykonávat funkce poskytované systémem, viz modely spolupráce. Operace určující chování jsou definovány a rozpoznávány svojí signaturou – názvem, seznamem parametrů a návratových hodnot. Objekt přijme zprávu a vykoná operaci, jejíž signatura je shodná se signaturou zprávy.

Pro lepší pochopení myšlenky OODB: http://www.linuxexpres.cz/business/objektove-databaze

Objektové databáze

Stejně tak jako lidé postoupili od strukturálního programování k objektovému, tak si řekli, že by nebylo od věci neukládat data do tabulek a relací, ale do objektů tak, jak s nimi pracujeme přímo v programu. Bylo by přeci velice pěkné, když bych mohl objekt tak, jak ho mám, prostě uložit do databáze a o nic víc se nemuset starat.
Nemusel bych přemýšlet nad strukturami tabulek (tak aby dodržovaly „dobré mravy“ dané normami) a tvořit ruční INSERTy a SELECTy, jen bych databázovému stroji přes nějaké API řekl, načti mi uživatele s číslem 451, a dostal bych ho se všemi atributy naplněnými.
A přesně takto objektové databáze fungují. Místo tabulek jsou zde uloženy přímo objekty, včetně svých vlastností, a místo řádků se ukládají samotné instance objektů. Každý takto vložený objekt je jednoznačně identifikován svým OID, které na logické úrovni odpovídá ukazateli do virtuální paměti počítače a stejně tak se chová (při přesunu v paměti se změní i OID). Není tedy potřeba vytvářet primární klíče na objektech ani normalizovat databázi.
Objektové databáze také nabízejí využití možností vícenásobné dědičnosti, zapouzdření a polymorfizmu. Navíc vlastnosti (datové hodnoty) objektů nemusí být jen primitivního typu, ale mohou být dále strukturované jako například objekt (pomocí reference), množina nebo seznam. Pojem zapouzdření znamená, že každý objekt obsahuje nejen datové hodnoty (vlastnosti), ale i funkce, které definují, jak je možné s těmito vlastnostmi zacházet.
Polymorfizmus umožňuje objektům zastupovat své potomky (ve smyslu dědičnosti) při volání metod. Program nemusí znát přesný typ objektu, který volá, ale ten se zjistí až za běhu a zavolá se metoda na správné třídě.
Pro vytváření nových tříd v databázi je definován nový speciální jazyk ODL (Object Definition Language). Nicméně v dnešní době se využívá vlastností pokročilých programovacích jazyků, jako je reflexe. Například v db4o stačí zavolat metodu set na objektu databáze, předat jí jako parametr obyčejný objekt a zbytek už si zařídí knihovna sama.

```
void storePilot (string pilotName, int pilotsPoints)
{
  Pilot pilot1 = new Pilot(pilotName, pilotsPoints);
  db.Set(pilot1);
}
```

Pro načítání dat z objektové databáze existuje jazyk OQL (Object Query Language). Jak je vidět už podle názvu, jeho syntaxe je velice blízká SQL. V následujícím příkladu si povšimněte, že odpadla potřeba spojovat tabulky, protože vše je dostupné přes objektové vazby:

```
SELECT o.customer_id.get_name(), o.room_id.id
FROM orders o
```

```
WHERE o.check_day(o.room_id.id, datefrom, dateto) = 1;
```

Další velice zajímavou možností je vytvořit dotaz pomocí QBE (Query By Example). Princip tohoto přístupu spočívá v tom, že databázi předáme částečně naplněný objekt a ta nám ho dohledá ve svém zdroji a doplní ostatní vlastnosti. Přesněji řečeno vrátí nějaký kontejner naplněný objekty, které původnímu objektu odpovídají. Uvedu zde jeden ilustrační příklad:

```
Pilot retrievePilotByName (string pilotName)
{
 Pilot proto = new Pilot("Michael Schumacher", 0);
 IObjectSet result = db.Get(proto);
 if (result.HasNext())
   return (Pilot)result.Next();
 else
   return null;
}
```

From <http://www.linuxexpres.cz/business/objektove-databaze>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

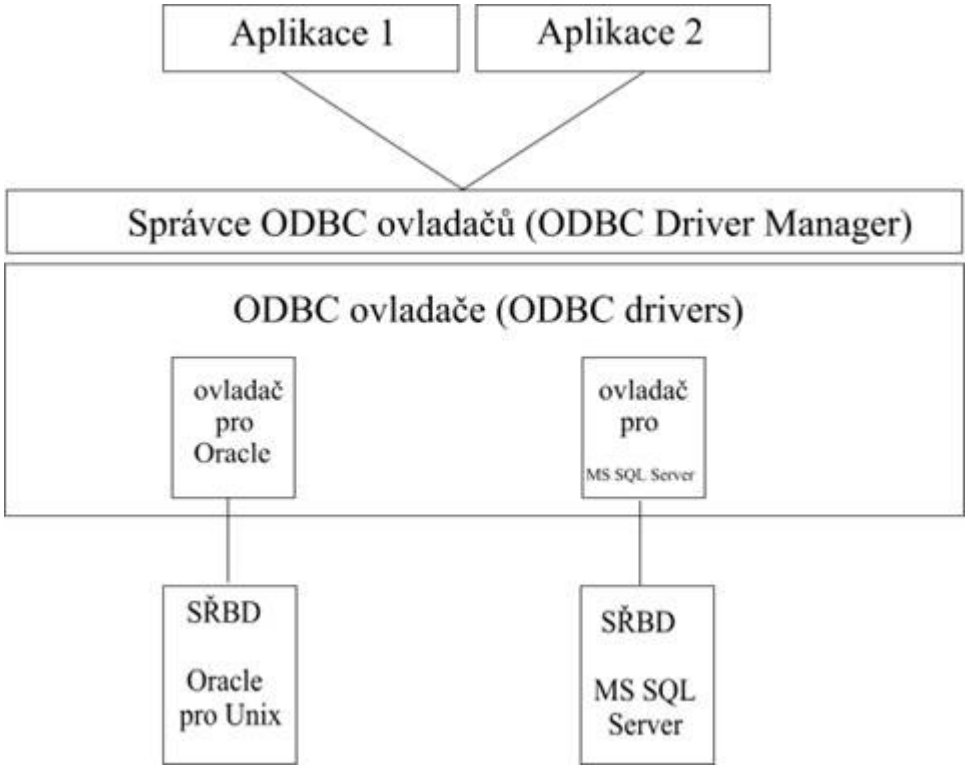
„Vnější“ programování (přes rozhraní/knihovny) – rozhraní ODBC, JDBC, rozhraní podporující objektově-relační mapování (Java Hibernate).

Thursday, May 30, 2013 8:20 AM

ODBC

Open DataBase Connectivity .Je standardizované softwarové API pro přístup k databázovým systémům (DBMS). Snahou ODBC je poskytovat přístup nezávislý na programovacím jazyku, operačním systému a databázovém systému. Je to čistě C-čkové API, které nemá žádný objektový základ.
Navrženo Microsoftem, proto primárně přístupné pouze přes C/C++. Založeno na specifikaci X/Open a ISO: SQL Call Level Interface (SQL/CLI)

Model struktury ODBC se dá znázornit pomocí čtyř vrstev:

- V první nejvrchnější vrstvě se nachází samotná aplikace. Ta v případě, že potřebuje data, provede volání ODBC funkcí (ve formě SQL dotazu).
- Druhou vrstvou je tzv. "Správce ODBC ovladačů" (ODBC Driver Manager). Úkolem správce ovladačů je zajistit propojení mezi aplikací a příslušným ODBC ovladačem (ODBC ovladače tvoří třetí vrstvu modelu, podrobněji viz dále). Jakmile aplikace potřebuje data, správce ovladačů vyhledá a nahraje příslušný ovladač. (ve formě DLL knihovny). Správce ovladačů také zjistí, jaké konkrétní funkce jsou podporovány jednotlivými ovladači, a uschová si jejich adresy v paměti do tabulky. V případě, že aplikace volá konkrétní funkci, správce souborů zjistí, ke kterému ovladači funkce patří a zavolá ji. Tímto způsobem může být prováděn souběžný přístup k více ovladačům, což se hodí v případě programování aplikací přistupujících souběžně k několika zdrojům dat.
- Třetí vrstvou zde již zmíněnou vrstvou jsou ODBC ovladače. Ty provedou zpracování volané ODBC funkce, přeložení požadavku do SQL pro příslušný SŘBD (DBMS) a jeho následné poslání.
- Poslední vrstvou je SŘBD, který provede zpracování operace požadované ODBC ovladačem a výsledky této operací mu vrátí.
- 

Open Database Connectivity (ODBC) is Microsoft's strategic interface for accessing data in a heterogeneous environment of relational and non- relational database management systems. Based on the Call Level Interface specification of the SQL Access Group, ODBC provides an open, vendor- neutral way of accessing data stored in a variety of proprietary personal computer, minicomputer, and mainframe databases.

ODBC alleviates the need for independent software vendors and corporate developers to learn multiple application programming interfaces. ODBC now provides a universal data access interface. With ODBC, application developers can allow an application t o

concurrently access, view, and modify data from multiple, diverse databases.

ODBC is a core component of Microsoft Windows Open Services Architecture. Apple has endorsed ODBC as a key enabling technology by announcing support into System 7 in the future. With growing industry support, ODBC is quickly emerging as an important industry standard for data access for both Windows and Macintosh applications.

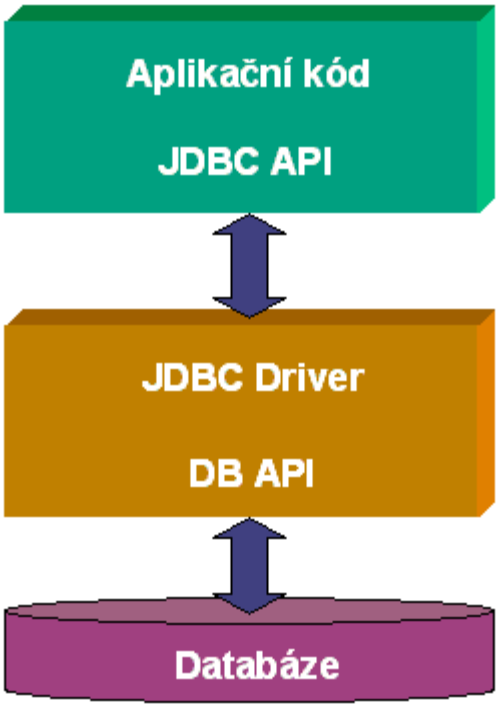
From <http://support.microsoft.com/kb/110093>

JDBC (Java Database Connectivity)

- jeho API poskytuje základní rozhraní pro unifikovaný přístup k databázím, aplikační programátor je tak odstíněn od specifického API databáze a může se naučit pouze jednotné rozhraní JDBC
- lze použít i mimo databáze – pro přístup k datům ve forme tabulek (CSV, XLS, ...)
- ovladače jsou k dispozici pro většinu databázových systémů

inspirováno rozhraním ODBC:

- objektové rozhraní
- strukturovanější a přehlednější
- možnost spolupráce s ODBC



JDBC ovladač

- zprostředkovává komunikaci aplikace s konkrétním typem databáze
- implementován obvykle výrobcem databáze
- dotazovací jazyk – SQL
  - předá se databázi
  - ovladač vyhodnotí přímo
- reprezentován specifickou třídou
  - sun.jdbc.odbc.JdbcOdbcDriver
  - com.mysql.jdbc.Driver

Typy JDBC ovladačů

Typ 1:

- využívá ODBC (pres JDBC-ODBC bridge)

- Obtížně konfigurovatelné

Typ 2:

- komunikace s nativním ovladačem nainstalovaným na počítači

Typ 3:

- komunikuje s centrálním serverem (Network Server) sítovým protokolem
- pro rozsáhlé heterogenní systémy, velmi efektivní i díky poolingu připojení

Typ 4:

- založen ciste na jazyce Java
- prímý prístup do databáze

Java Hibernate

Hibernate je framework napsaný v jazyce Java, který umožňuje tzv. ORM – Objektově-Relační mapování. Usnadňuje řešení otázky zachování dat z objektů i po ukončení běhu aplikace. Provádí podobné věci jako např. JPA – Java Persistence API.

Co dělá hibernate

Hibernate poskytuje způsob, pomocí nějž je možné zachovat stav objektů mezi dvěma spuštěními aplikacemi. Říkáme tedy, že udržuje data persistentní. Dosahuje toho pomocí ORM, což znamená, že mapuje Javovské objekty na entity v relační databázi. K tomu používá tzv. mapovací soubory, ve kterých je popsáno, jakým způsobem se mají data z objektu transformovat do databáze a naopak, jakým způsobem se z databázových tabulek mají vytvořit objekty. Druhý způsob, jak mapovat objekty, je použít anotace místo mapovacích souborů. V Hibernate tedy pracujete se svými normálními business objekty, pouze pro každý atribut přidáte get/set metody a metody hashCode() a equals(). Nutno podotknout, že nelze použít EJB(viz.Java Bean), ale pouze tzv. POJO(Plain Old Java Object). Poté, co máte objekty uložené v databázi se na ně můžete dotazovat jazykem HQL (Hibernate Query Language), který je odvozen z SQL a je mu tedy velice podobný.

Objektovou struktura mého programu předhodím Hibernatu a základě tohoto objektového modelu (a označením objektů které tam požaduju) si Hibernate vytvoří vlastní schéma, aby věděl kde jsou data uložena.

Výhody používání Hibernate

Hibernate, framework pro perzistentní vrstvu, usnadňuje programátorovi práci tím, že nemusí transformovat objekty do relací ručně, ale přenechá to perzistentní vrstvě. Zároveň jsou tím odstíněna specifika jednotlivých databází – programátor používá API Hibernate.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Objektově relační mapování

S relačními databázemi přichází i potřeba podpory **objektově relačního mapování**, což je proces překladu objektů na tabulkovou reprezentaci a překlad vazeb a vztahů mezi těmito objekty do dodatečných tabulek

Vloženo z <http://www.cs.vsb.cz/behalek/frvs/2005/java/hibernate/hibernate.html>

Distribuované databáze – koncepce distribuovaného databázového systému, replikace a fragmentace dat, distribuovaná správa transakcí.

Thursday, May 30, 2013 8:20 AM

Distribuovaná DB je množina databází, která je uložena na několika počítačích. Uživateli se může jevit jako jedna velká databáze. V DB neexistuje žádný centrální uzel nebo proces odpovědný za vrcholové řízení funkcí celého systému. Výrazně to zvyšuje odolnost systému proti výpadkům jeho částí. Data i funkce rozděleny mezi více počítačů

Charakteristické vlastnosti

Distribuovaná DB je charakterizována:

- Transparentností – z pohledu klienta se zdá, že data jsou zpracována na jednom serveru v lokální databázi. Uživatel si rozdělení nemusí být vědom.
- Rozšiřitelností – zvýšení výkonu přidáním dalších počítačů.
- Robustností – výpadek jednoho počítače neovlivní funkci ostatních
- Jsou syntakticky shodné příkazy pro lokální i vzdálená data, nespecifikuje se místo uložení dat (řeší to distribuovaný SŘBD)
- Autonomností – s každou lokální bází dat zapojenou do distribuované databáze je možno pracovat nezávisle na ostatních databázích.
- Lokální Db je funkčně samostatná, připojení do jiné části distribuované db se v případě potřeby zřizují dynamicky.
- Nezávislost na počítačové síti – jsou podporovány různé typy architektur lokálních i globálních počítačových sítí (LAN, WAN)
- V distribuované databázi mohou být zapojeny počítače i počítačové sítě různých architektur, pro komunikaci se používá jazyk SQL.

Distribuované databáze jsou výhodné kvůli:

- Lokální autonomie – odpovídají struktuře decentralizovaných organizací. Data jsou uložena v místě nejčastějšího využití a zpracování – zlevnění provozu
- Zvýšení výkonu (rozdělení zátěže na více počítačů)
- Spolehlivosti (replikace dat, degradace služeb při výpadku uzlu, přesunutí na jiný uzel)
- Rozšiřitelnosti
- Schopnosti sdílet informace integrací podnikových zdrojů
- Agregaci informací – z více bází dat lze získat informace nového typu.

Naopak Distribuované Db mohou způsobit i pro ně specifické problémy:

- Složitost – distribuce db, distribuce zpracování dotazu a jeho optimalizace, složité globální transakční zpracování, distribuce katalogu, paralelismus a uvíznutí, složité zotavování z chyb
- Cena (komunikace je navíc)
- Bezpečnost
- Obtížný přechod – neexistuje automatický konverzní prostředek z centralizovaných DB na DDB

Taxonomie DDBS

- **Těsně integrované**
  - Uživatel vidí data centralizovaná v jediné databázi, DDB je vybudována nad lokálními DB, každé místo má úplnou znalost o datech v celém DDBS a může zpracovávat požadavky používající data z různých míst.
- **Semiautonomní**
  - Lokální DBMS pracují nezávisle a sdílejí svoje lokální data v celé federaci, jen část jejich dat je sdílena.

- **Zcela autonomní**
    - Izolované, lokální DBMS pracují nezávisle a neví o ostatních DBMS, pro vzájemnou komunikaci potřebují softwarovou vrstvu pracující nad jednotlivými DBMS.

Problémy replikace a fragmentace dat

U fragmentace a replikace bychom měli respektovat hlediska:

- Rozdělit relace do lokálních serverů tak, aby aplikace zatěžovaly servery stejnoměrně.
- Přístupnost a spolehlivost - Replikací zlepšíme spolehlivost a read-only dostupnost.
- Lokality zpracování (maximalizovat lokální).
- Dostupnosti a ceny paměti v jednotlivých uzlech

Replikace dat

Replikační transparentnost je neobtěžovat uživatele skutečností, že pracuje s daty existujícími ve více kopiích = uživatel neví o replikách.

Při replikaci dat se nachází kopie množiny objektů v každém uzlu, ve kterém je využívaná. V systému tedy existuje několik kopií každého objektu. Výhodou tohoto řešení je kvalitní dostupnost, rychlý přístup ke každému objektu a menší nároky na komunikaci mezi uzly. Nevýhodou je problém duplicity, díky níž je nutné trvale *zajišťovat konzistenci všech kopií*. Je nutné implementovat systém, který určí správné pořadí provedených operací a při replikaci rozdistribuuje správnou kopii. Také je nutné zabránit současné modifikaci dvou kopií objektu.

Správné pořadí provedených operací je určováno většinou časovými razítky, současné modifikaci dvou kopií jednoho objektu mohou zabránit klasické paralelní synchronizační prostředky (zámky, semafory apod.)

Jsou dva základní způsoby zacházení s replikami:

1. *Pouze pro čtení (master-slave)* - změny (zápis) může provádět jen master, repliky jsou read-only a periodicky se synchronizují s masterem; repliky tak mají lehce zastaralá, nekonzistentní data (tzv. eventual consistency, nakonec se dojde do konzistentního stavu)
2. *Pro transakční zpracování (multi-master)* - rovnocennější, změny možné provádět v replikách, je proto nutná obousměrná synchronizace, mohou tak vznikat konflikty, které se řeší různě (priority, časová razítka, manuálně, vlastní procedura)

Jiný zdroj:
Replikace = uchování kopií relací v různých uzlech.
výhody: porucha v jednom uzlu neznemožní přístup k jeho lokálním relacím, data v lokálních bázích jsou připravena k použití okamžitě, bez nutnosti přenosu
nevýhody: ztížení aktualizace, všechny kopie musí být aktualizovány současně a v průběhu aktualizace je nutno uzamknout aktualizovaná data ve všech uzlech sítě.
Proto se replikují data, ke kterým je potřebný rychlý přístup a která nejsou často aktualizována, příp. která jsou pro systém mimořádně důležitá (číselníky, registry ap.)
[http://barborka.vsb.cz/prednasky/presentations/2006-DAIS-vp-Databazove_a_informacni_systemy/Prezentace/Dais_7.pdf]

Fragmentace dat

Fragmentační transparentnost = uživatelův dotaz je specifikován na celou relaci, ale musí být vykonán na jejím fragmentu = uživatel neví o fragmentech.

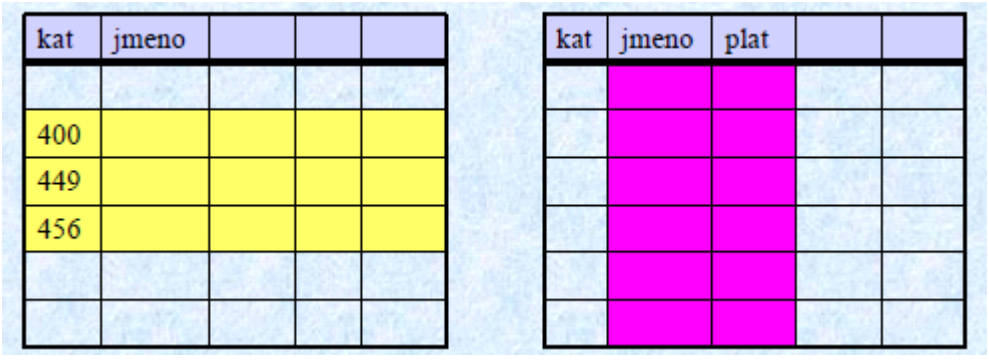
Fragmentace dat se dělí na:

- Horizontální – dle selekční podmínky rozdělíme tabulku na 2 části horizontálním řezem, tedy např. na knihy s ISBN nižším než 122 a na knihy s ISBN větším nebo rovným 122.
- Odvozená horizontální – dochází k rozdělení tabulky na 2 a více částí horizontálním řezem, v tomto případě však je fragmentace založena na jiné relaci. Např. DODAVATELE a KNIHY. DODAVATELE rozdělíme do fragmentů a na základě těchto fragmentů provedeme fragmentaci v tabulce KNIHY, která je spojena s DODAVATLE relací.

- Vertikální – rozdělení tabulky podle sloupců na 2 a více částí. V jedné skupině jsou jedny sloupce, v druhé jiné.
- Smíšená – tabulku např. rozdělíme dle sloupců (vertikální) a následně v jednotlivých částech provedeme horizontální fragmentaci (a nebo obráceně).

Jiný zdroj:

Fragmentace = rozložení relace na části (fragmenty), které jsou umístěny v různých uzlech sítě.
Může jít o horizontální fragmentaci, kdy se v různých uzlech ukládají části relace rozložené do skupin řádků, nebo o vertikální fragmentaci, kdy se v uzlech ukládají různé projekce relace. Fragmentace se provádí tak, aby bylo možno z fragmentů získat původní relaci standardními operacemi nad relační databází (sjednocením nebo spojením).



kat	jmeno			
400				
449				
456				

kat	jmeno	plat		

Distribuovaná správa transakcí

- Pro provedení transakční změny dat ve více databázích najednou
- K tomu se využívá **dvoufázový commit** (2PC)
  - Jde o protokol řešení distribuovaných transakcí
  - Řídící uzel (koordinátor) řídí průběh celé distribuované transakce
  - Ostatní zainteresované uzly poslouchají

1. **Fáze PREPARE**

  - **Koordinátor** zašle zprávu PREPARE všem uzlům účastnících se distrib. transakce
  - Koordinátor v této fázi nic nedělá, čeká jen na výsledky od ostatních uzlů
  - **Každý podřízený uzel** se pokusí provést fázi PREPARE
    - Neprovádí vlastní commit, transakce je pořád neukončená, změněné řádky zůstavají stále zamčené
    - Jen se zajistí, aby aktuální stav rozpracované transakce byl schopný přežít výpadek
    - Každý uzel odpoví zpět koordinátorovi PREPARED (pokud vše ok) nebo ABORT (pokud nastal problém)

2. **Fáze COMMIT**

  - Pokud obdržel koordinátor od všech uzlů PREPARED, provede potvrzení svých lokálních změn (normální commit)
  - A všem uzlům zašle zprávu COMMIT
  - Pokud je aspoň 1 výsledek ABORT, provede ROLLBACK svých změn (standardní COMMIT) + všem uzlům zašle zprávu ROLLBACK

- Protokol je imunní vůči výpadku sítě/uzlu v libovolné fázi
- Koordinátor si po celou dobu zpracování globální transakce udržuje data o stavu transakce a všech podtransakcích
  - Je tedy schopen transakci dokončit např. při obnovení spojení se vzdáleným uzlem

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Temporální databáze, porovnání klasických a temporálních databází, modely času, vztah událostí a času (snapshot), temporální SQL.

Thursday, May 30, 2013 8:20 AM

Temporální databáze jsou databáze určitým způsobem podporujícím čas. Čas potřebujeme v databázích např. ve studijním informačním systému, účetních a bankovních systémech, docházkových systémech. Hlavní cíl temporálního DM by měl být zachytit sémantiku dat měnící se v čase. V praxi existuje mnoho nekompatibilních datových modelů s mnoha dotazovacími jazyky.

Temporální databáze (temporální databázový systém) je databáze (databázový systém) zohledňující časové vlastnosti ukládaných dat

Jméno	Plat	Funkce	Datum narození	Platí_od	Platí_do
Pepa	60000	Vrátný	1945-04-09	1995-01-01	1995-06-01
Pepa	70000	Vrátný	1945-04-09	1995-06-01	1995-10-01
Pepa	70000	Vrchní vrátný	1945-04-09	1995-10-01	1995-02-01
Pepa	70000	Ředitel bezpečnosti	1945-04-09	1996-02-01	1997-01-01

Porovnání klasických a temporálních DB

Klasický DB systém

Zachycuje stav systému v aktuálním časovém okamžiku. Problém: co dělat se starými daty. V případě, že se systém v čase vyvíjí, změny se v DB projeví přidáním nových informací a mazáním starých. Klasické DB neobsahují informaci o čase. V případě, že požadujeme uchování historie změn, či alespoň předchozího stavu, je nutné do DB doplnit informace o čase. Aktualizaci a operace s časem musí zajistit uživatel, což není triviální.

Temporální DB

Databáze určitým způsobem podporující čas. Poskytuje vhodný dotazovací jazyk zahrnující práci s časem – výhodou jsou jednodušší dotazy, v nichž se vyskytuje čas, což přináší méně chyb v aplikačním kódu a zajišťuje jednodušší udržování aplikací.

Temporální projekce – jako projekce v klasických databázích (z celé relace jsou vybrány hodnoty podle zadaných atributů), navíc bere v úvahu čas. V případě, že dvě n-tice výsledku mají stejné hodnoty všech svých atributů a překrývají se nebo dotýkají se časem, srostou tyto dvě n-tice s časem odpovídajícím sjednocení obou n-tic.

Temporální spojení – stejné jako spojení (JOIN) v klasických DB (zadáme sloupce a podmínku, která říká, kdy jsou dva řádky tabulky spojeny), navíc bere v úvahu čas události. V Temporálních DB nemusíme zadávat sloupce uchovávající čas, pouze podmínku na spojení pro čas.

Jiný zdroj:

Klasické databáze

- Neobsahují informaci o čase
- V databázi zachycen pouze aktuální stav systému. V případě, že se v čase systém vyvíjí, změny se v databázi projeví přidáváním nových informací a mazáním starých.
- V případě, že požadujeme uchovávání historie změn, či alespoň předchozího stavu, je nutné do databáze doplnit informaci o čase. Aktualizaci a operace s časem musí zajistit uživatel. Což (jak ilustrujeme dále) není triviální.

- Jako příklad poslouží SIS, kde chceme uchovávat informace o předcházejících semestrech.
  Řešením je přidání sloupce, který identifikuje konkrétní semestr. Nevýhodou tohoto řešení je, že s touto informací musí manipulovat uživatel sám.

Temporální databáze

- Konkrétní podporu času uvidíme později
- Vhodný dotazovací jazyk zahrnující práci s časem
- Výhodou jsou jednodušší dotazy v nichž se vyskytuje čas, což přináší méně chyb v aplikačním kódu

Modely času

Temporální logika: čas je libovolná množina okamžiků s daným uspořádáním. Modely času se rozlišují podle:

Dle uspořádání

- **Lineární** – čas roste od minulosti k budoucnosti lineárně
- **Větvený** (čas možných budoucností) – lineární minulost až do teď, pak se větví do několika časových linií reprezentujících možný sled událostí. Každá linie se může dále větvit.
- **Cyklický** – opakující se procesy. Př. týden, každý den se opakuje po sedmi dnech.

Dle hustoty

- **Diskrétní** – spolu s lineárním uspořádáním. Každý okamžik má právě jednoho následníka.
- **Hustý** – Mezi každými dvěma okamžiky existuje nějaký další
- **Spojitý** – každé reálné číslo odpovídá bodu v čase.

– Omezenost času – Omezený – nutnost zejména kvůli reprezentaci v počítači, Neomezený.
– Absolutní /relativní čas – Absolutní se vyjádří hodnotou, také ale potřebuje počátek. Relativní vyžaduje nějaký počátek, čas se pak vyjádří jako vzdálenost a směr od počátku.

Datové typy pro čas jsou:

- Časový okamžik (instant) – DATE, TIME, TIMESTAMP
- Časový úsek (time period) – doba mezi dvěma časovými okamžiky (15:30-16:00)
- Časový interval (interval) – doba o specifikované délce, ale bez konkrétních krajních bodů (30 minut)
- Množina časových okamžiků (instant set)
- Množina časových úseků (temporal elements)

Vztah událostí a času (snapshot)

Čas platnosti (valid time)

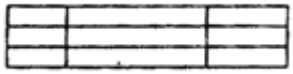
- Čas, kdy byla událost pravdivá v reálném světě. Může být v minulosti přítomnosti i budoucnosti.
- Reprezentace času platnosti – časový okamžik, doba, časový úsek, množina okamžiků
- Čas platnosti může být přidružen k atributům, množině atributů, celé n-tici nebo objektu

Transakční čas (transaction time)

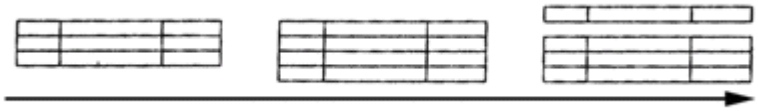
- Čas, kdy byl fakt reprezentován v DB. Nabývá pouze aktuální hodnoty. Monotónně roste.
- Reprezentace transakčního času – časový okamžik (nová n-tice se stejným klíčem – logické odstranění původní), časový úsek (teď, dokud nezměněno), tři časové okamžiky (čas zaznamenání začátku v reálném světě, konce události v reálném světě, logického odstranění události z db), množina časových úseků

Snapshot

Datový model nepodporující čas platnosti ani transakční čas. Je to klasický relační model. Každá n-tice je fakt platný v reálném světě. Při změně reálného světa jsou do relace prvky přidávány nebo z ní odebírány.

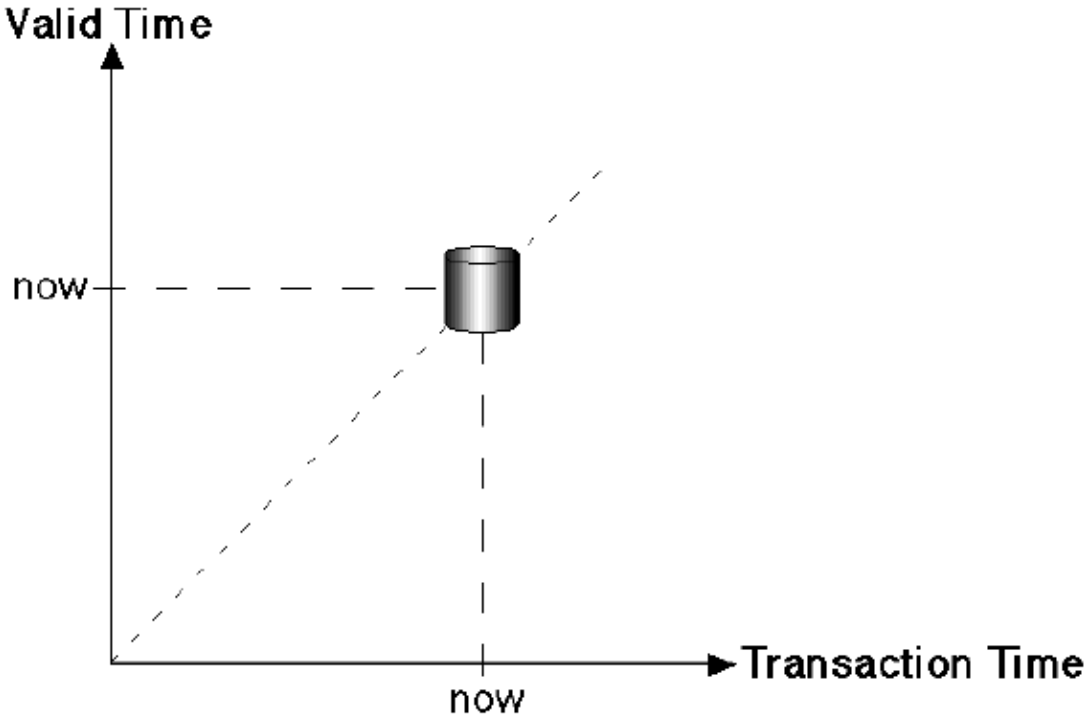


Další datové modely mohou být **valid-time** relace (podporuje čas platnosti, umožňuje klást dotazy o faktech z minulosti i budoucnosti), **transaction-time** relace (podporuje pouze transakční čas, umožňuje získat informaci ze stavu db v nějakém okamžiku v minulosti), bitemporální, temporální.

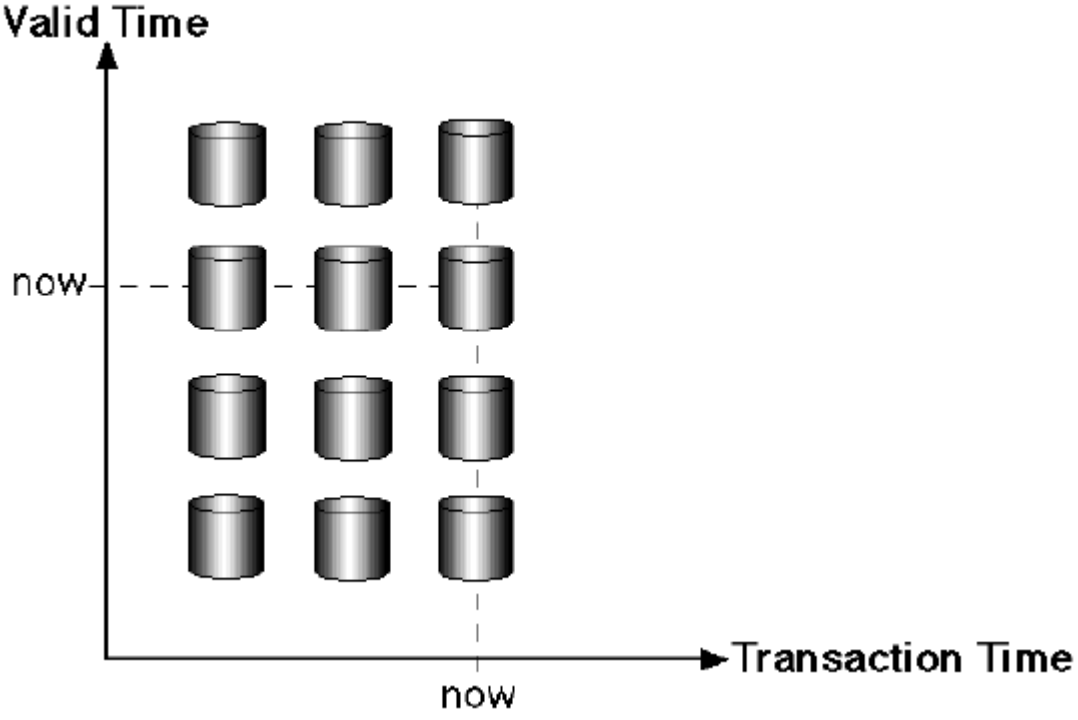


Obr – transaction-time

Snapshot database:



Temporal database:



Temporální SQL

Temporální datový model obsahuje objekty s přesně danou strukturou, omezení pro dané objekty a operace na daných objektech (temporální dotazovací jazyky). Temporálních dotazovacích jazyků je velké množství. Nejčastěji jsou založené na SQL. Typy jsou:

- Relační – HQL, HSQL, TDM, TQuel, TSQL, TSQL2
- Objektově orientované – Matisse, OSQL, OQL, TMQL

TSQL2

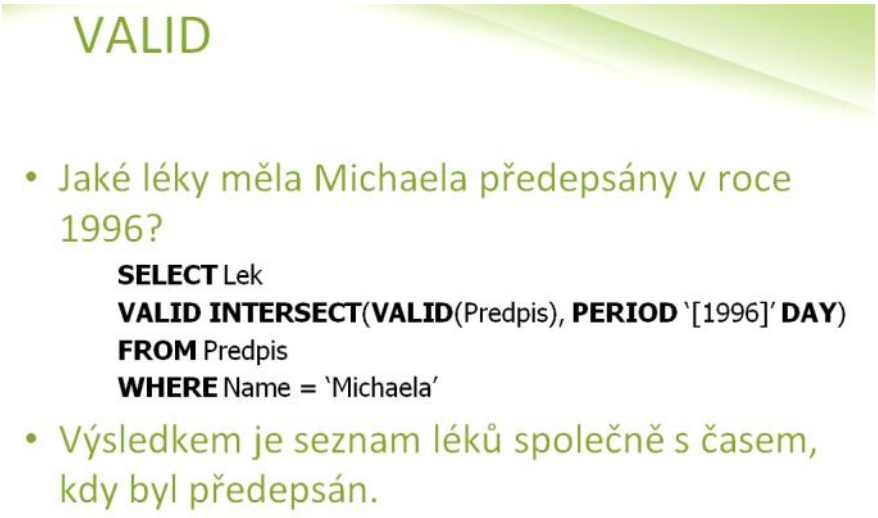
Temporal SQL 2 – měl sjednotit přístupy k temporálním datovým modelům. Je to nadmnožina SQL92.

V TSQL2 je časová osa na obou koncích omezena, ale dostatečně daleko (18 miliard let). U časových údajů jsou možné různé granularity. Časové typy: DATE, TIME, TIMESTAMP, INTERVAL, PERIOD.

Datový model je bitemporální. Řádek je orazítkován množinou bitemporálních chrononů. Bitemporální chronon je dvojice (chronon transakčního času, chronon času platnosti). Př. Relace ZAMESTNANEC – umístění lidí v odděleních určitého podniku. Schéma (Jméno, Oddělení) + časové razítko.

Př. SELECT – komu byl předepsán nějaký lék – výsledek bez podpory času

```
SELECT SNAPSHOT Jmeno FROM Predpis
```



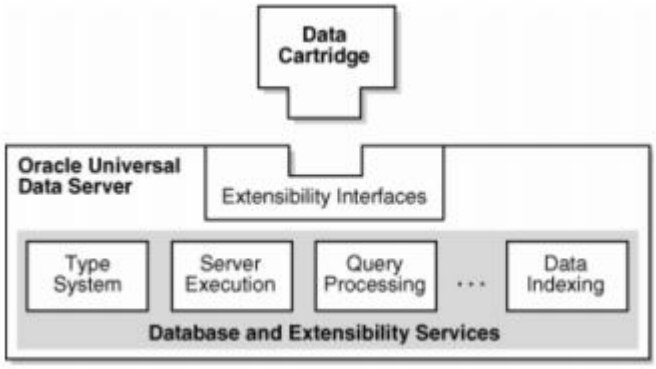
From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Uživatelské rozšíření databázových systémů – data cartridge, příklady použití.

Thursday, May 30, 2013 8:21 AM

- rozšiřitelnost = možnost přidávání datových typů a programů (funkcí) zabalených do speciálního modulu
- uživatelsky definované
  - typy = UDT
  - funkce = UDF
- problém: zapojení do relačního SŘDB včetně SQL
  - DB/2 = relační extendery
  - Informix = DataBlades
  - Sybase = Component Integration Layer
  - Oracle = cartridges

Oracle Extensibility Framework



- Data cartridge
  - způsob uživatelského rozšíření databázového serveru Oracle
  - rozšíření je pouze na straně serveru
  - jsou integrované se serverem přes rozhraní
  - jsou v podobně balíků = instalují se jako celek
  - http://docs.oracle.com/cd/B10500_01/appdev.920/a96595/dci01wht.htm
- Extensibility interface
  - rozhraní, které umožňuje vytvářet cartridge jednotným způsobem
  - poskytuje přesně definovaný způsob, jak se serverem komunikovat
  - zpřístupňuje jednotlivé standardní služby, které lze v serveru rozšířit
- Database and extensibility services
  - sada standardních služeb, které poskytuje server
  - lze je využívat a rozšiřovat pomocí cartridge
  - jednotlivé služby jsou:
    - Extensible type system
      - podpora pro uživatelské typy, kolekce, reference (REF), LOBy
    - ExtensibleServer execution environment
      - podpora vlastních procedur a funkcí
    - Extensible Indexing
      - vlastní způsob indexování = tzv. doménové indexování pro doménově specifická data
    - Extensible Optimiser
      - podpora pro vytváření uživatelských funkcí a indexů pro vlastní sběr statistik pro CBO

- Pracuje na principu tzv. cartridge
  - způsob uživatelského rozšíření databáze Oracle
    - integruje se pomocí sady rozhraní pro jednotlivé přístupy (k datům, indexům,…)
  - rozšíření může definovat
    - nové datové typy(standardní, pole, hnízděné tabulky, LOBy) a jejich funkce
    - nové typy indexů
    - nové operátory
  - proč implementovat DataCartridge
    - nutnost zpracování komplexních dat, která neodpovídají standardním relačním informacím
    - nutnost snadné manipulace s takovými daty
  - Příklady
    - multioborové = datové, statistické výpočty, prostorové databáze, multimédia
    - specializované, finanční a právní systémy
- typická struktura cartridge
  - definice nových objektových typů
  - implementace těl typů, balíky, procedury, funkce
  - případné DLL knihovny s implementací v C
  - operátory
  - doménové indexy pro podporu operátorů
- př. standardní cartridge: podpora indexování a vyhledávání v textech

Př. cartrige:

LOB - Large Objects

- standardní typy pro ukládání objemných dat na serveru (až 4 GB)
- typy
  - Externí
    - BFILE = samostatný binární soubor uložený vně databáze
  - Interní
    - CLOB = znakový typ
    - NCLOB = znakový typ v národní sadě
    - BLOB = binární typ

Oracle - Define CLOB and NCLOB datatypes - June 27, 2009 at 11:00 AM
Define CLOB and NCLOB datatypes.
Both CLOB and NCLOB are used to store huge character data in the database.
CLOBs store single-byte character set data.
NCLOBs store fixes-length multi-byte character set data.
Both these datatypes participate fully in transactions.

From <http://www.careerride.com/Oracle-CLOB-and-NCLOB-datatypes.aspx>

- ve sloupci tabulky uložen pouze deskriptor odkazující na samotná data
  - hodnoty pro xLOB slupce = NULL, EMPTY_CLOB(), EMPTY_BLOB(), EMPTY_NCLOB()
- manipulace
  - Oracle nabízí balík DBMS_LOB s řadou funkcí a procedur pro standardní manipulaci s daty
  - provádí se po částech pomocí bufferů
- indexace
  - není standardně indexováno, ale je možné implementovat svá vlastní rozhraní pro indexaci

Rozšíření serveru

- server dovoluje psát implementace procedur v řadě jazyků
  - nativním PL/SQL
  - Javě
  - v čemkoliv s konvencí jazyka C a kompilací DLL

http://download.oracle.com/docs/cd/B14117_01/appdev.101/b10800/dciwhatis.htm

Příklady rozšíření

- Matematické, finanční funkce
- Práce s audio, video objekty v reálném čase
- GIS a prostorové objekty
- Textová rozšíření (různé statistiky textu)

Cartridge Oracle Text (z nadpisu jeji prednasky, takze to bylo asi dulezite)

Oracle Text and Ultra Search
Oracle Text brings search engine-like full text search capabilities to the Oracle Database. Ultra Search provides a ready-to-use application, while Oracle Text provides a foundation for building your own search applications. Search regular columns, text inside various kinds of binary-format documents, and text, tags, and attributes inside XML documents.

From <http://www.oracle.com/pls/db111/portal.portal_db?selected=7>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Dokumentografické systémy, fulltextové vyhledávání, filtrace, disambiguace, lemmatizace, indexy, tezaury, dotazování.

Thursday, May 30, 2013 8:21 AM

Dokumentografické systémy (DIS)

- vznik 50. léta 20. stol. za účelem automatizace postupů používaných v knihovnictví
- Nyní samostatná podčást IS
  - Faktografický IS - informace s definovanou vnitřní strukturou (nejčastěji tabulky)
  - Dokumentografický IS - informace v podobě textu v přirozeném jazyce bez pevné vnitřní struktury

Práce s DIS:

- *Zadání dotazu*
- *Porovnání*
- *Získání seznamu odpovídajících dokumentů*
- *Ladění dotazu*
- *Vyžádání dokumentu*
- *Obdržení textu*

Struktura DIS:

- Systém zpřístupnění dokumentu - sekundární informace o dokumentu (Autor, Název, ...)
- Systém dodání dokumentu - někdy není řešeno pomocí SW

Vyhodnocení dotazu

- přímé porovnávání náročné na čas

Ale při využití indexace a modelu dokumentu:

- nutné vytvořit model dokumentu
- ztrátový proces, založený na identifikaci slov v dokumentech
- výsledkem strukturovaná data vhodná pro porovnávání
- dotaz se upraví do odpovídající podoby a porovná se s modelem dokumentů

Text

Předzpracování

- vyhledávání nad modelem efektivnější, ale lze použít jen informace z modelu
- cíl: vytvořit model, zachovávající nejvíce info z původního textu
- Problém: nejednoznačností (ambiguity)
- dosud neřešené nároky na encyklopedické i asociativní znalosti

Porozumění textu

- Homonymie slov
  - jedno slovo může mít stejný tvar pro různé pády a další gram. jevy
    - kontroly: 1.p.m.č., 2.p.j.č. - není zřejmé jestli více kontrol nebo jedna kontrola
  - jeden tvar může mít různý význam
    - hnát - sloveso, podst. jm.
    - pět - číslovka, sloveso
- přiřazení je závislé na osobě, která dokument píše nebo čte
  - dva lidé mohou jednomu slovu přikládat zcela nebo částečně jiný význam
  - dva lidé si i pod stejným významem mohou představit jiný konkrétní předmět nebo množinu
    - máma, pokoj, ...
- výsledkem situace, kdy dva různí čtenáři nemusí přečtením získat stejnou informaci jako autor, ani

navzájem.
- Homonymie a nejednoznačnosti narůstají při přechodu od slov k větám.

Přesnost a úplnost

- důsledek nejednoznačnosti: žádný existující DIS nedává ideální výsledky
- Pro zobrazení odpovědi na dotaz lze určit
  - N_v (počet vrácených dokumentů) - O nich si DB myslí, že jsou relevantní, odpovídající dotazu
  - N_{vr} (počet vrácených relevantních dok.) - o nich si tazatel myslí, že uspokojí jeho požadavky
  - N_r (počet všech relevantních dok. v DB) - problematické u velkých DB
- Kvalita výsledné množiny se měří na základě:
  - Přesnost (Precision): $P = N_{vr} / N_v$
    - pravděpodobnost, že dokument zařazený v odpovědi je skutečně relevantní
  - Úplnost (Recall): $R = N_{vr} / N_r$
    - pravděpodobnost, že skutečně relevantní dokument je zařazený v odpovědi
- koeficienty jsou opět závislé na subjektivním názoru tazatele
- dokument vrácený na výstupu může uspokojovat požadavky dvou uživatelů, kteří položili stejný dotaz, různou měrou
- ideální případ: P == R == 1

Kritéria

- **Kritérium predikce**
  - při formulaci dotazů je třeba uhádnout, které termy (slova) byly v dokumentu autorem použity pro vyjádření dané myšlenky
    - problémy způsobují
      - synonyma - autor používá synonyma, které si tazatel nemusí při dotazu uvědomit
      - překrývající se význam slov
      - opisy jedné situace jinými slovy
  - částečné řešení - zařazení tezauru, který obsahuje
    - hierarchie slov a jejich významů
    - synonyma slov
    - asociace mezi slovy
  - tazatel může tezaurus využít při formulaci svých dotazů
  - při ladění dotazů má uživatel tendenci postupovat konzervativně
    - v dotazu často zůstávají ty části, které uživatele napadly na začátku a mění se jen podružné části, které nekvalitní výsledek nemusí zásadně ovlivnit
  - vhodné je uživateli pomoci s odstraněním nevhodných částí dotazu, které nepopisují relevantní dokumenty a naopak s přidáváním formulací, které relevantní dokumenty popisují
- **Kritérium maxima**
  - tazatel obvykle není schopen (ochoten) procházet příliš mnoho dokumentů do té míry, aby se rozhodl, zda jsou pro něj relevantní nebo ne
  - obvykle 20-50 podle velikosti
  - potřeba nejen dokumenty rozlišovat na odpovídající/neodpovídající dotazu, ale řadit je na výstupu podle míry předpokládané relevance
  - v důsledku kritéria maxima se při ladění dotazu uživatel obvykle snaží zvýšit přesnost
    - malé množství dokumentů v odpovědi, obsahující co největší poměr relevantních dokumentů
  - některé oblasti použití vyžadují co nejvyšší přesnost i úplnost
    - např. právnictví

<u>Modely dokumentografických systémů</u>

Úrovně modelů:

- rozlišují (ne)přítomnost slov v dokumentech
- rozlišují frekvence výskytů slov
- rozlišují pozice výskytů slov v dokumentech

Boolský model

- vznik 50. léta 20. stol., automatizace postupů používaných v knihovnictví
- Databáze obsahuje dokumenty, dokumenty popisovány pomocí termů, reprezentace dokumentu pomocí množiny termů (obsažených v dokumentu, popisujících význam dokumentu)
- **Indexace**
  - Přiřazení množiny termů, které jej popisují ke každému dokumentu
    - Ruční - nekonzistence
    - Automatická - konzistentní, ale bez porozumění textu
    - Řízená - předem daná množina termů
    - Neřízená - množina termů se mění s přibývajícími dokumenty
  - Tezaurus - vnitřně strukturovaná množina termů
    - Synonyma s preferovanými termy
    - Hierarchie užších/širších termů
    - Příbuzné termy
    - ...
  - Stop-list - nevýznamová slova
  - Příliš obecná slova nejsou pro identifikaci dokumentů vhodná, příliš specifická slova také ne
  - dotaz vyjádřen logickým výrazem: AND, OR, NOT
  - Příklad dotazu: počítač AND NOT osobní
  - Víceslovné termy: počítač AND NOT osobní počítač
  - Organizace indexu:
    - Invertovaný seznam - pro každý term je seznam dokumentů, ve kterých se vyskytuje
    - Zpracování dokumentů na vstupu - vznikne posloupnost dvojic <dok_id,term_id>
    - Setřídění dle term_id,dok_id
- **Nevýhody**
  - formulace dotazů je spíše uměním než vědou
  - nemožnost ohodnotit vhodnost vystupujících dokumentů
  - všechny termy v dotazu i v identifikaci dokumentu jsou chápány jako stejně důležité
  - nemožnost řízení velikosti výstupu
  - některé výsledky neodpovídají intuitivní představě

Vektorový model

- vznik 70. léta 20. stol., cca o 20 let mladší než Booleovské DIS
- snaha minimalizovat nebo odstranit nevýhody Booleovských DIS
- Struktura:
  - databáze obsahuje dokumenty
  - dokument popisován pomocí množiny termů
  - term je slovo nebo sousloví
  - reprezentace dokumentu pomocí vektoru vah termů

Tezaury

In Microsoft SQL Server 2005, full-text queries can use a thesaurus to find synonyms of search terms. For each supported language, there exists a single thesaurus file.

From <http://msdn.microsoft.com/en-us/library/ms142491(v=SQL.90).aspx>

Tezaurus je vnitřně strukturovaná množina termů:
-synonyma s preferovanými termy
-hierarchie užších/širších termů
-asociace mezi slovy

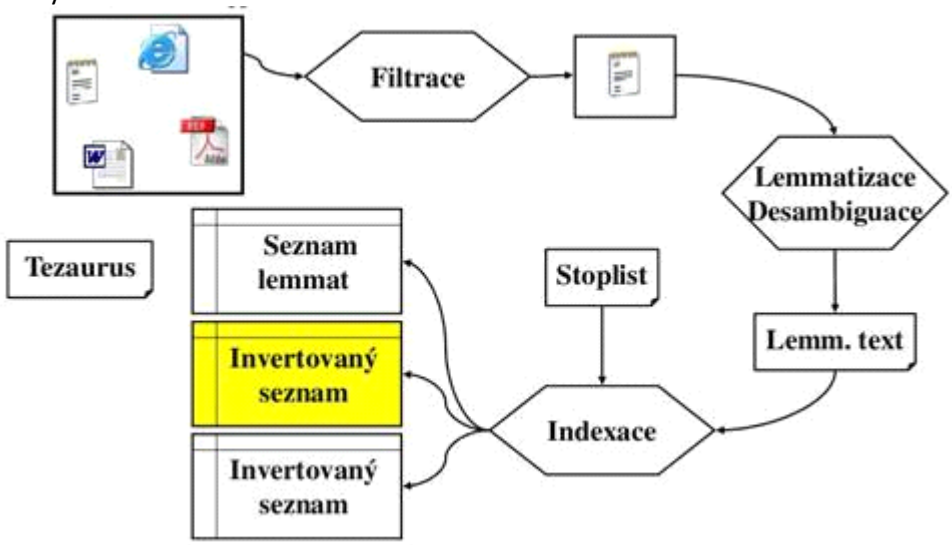
Fulltextové vyhledávání

- Odlišné od principů běžného vyhledávání
  - neprohledávají se striktně strukturovaná data, kde má každý sloupec každé tabulky předem

daný význam
    - prohledávají se volně psané texty, kde může být stejná událost popsána více autory rozdílně - různá slova stejného významu, různé slovní obraty a opisy
- DB systémy využívají svých prostředků rozšiřitelnosti a dodávají standardně prostředky, které vyhledávání v textových datech umožňují
- Rozdílné přístupy a možnosti
    - neexistuje objektivně nejlepší řešení
    - výsledky navíc podléhají subjektivním názorům tazatelů
- Samotná formulace dotazu, který by vrátil všechny dokumenty, které tazatele zajímají a žádné jiné, obvykle nelze zformulovat
    - spolu s vyhovujícími - relevantními - odpověďmi se obvykle vrací i odpovědi nerelevantní
- **Problémy**
    - Homonyma - ptá se tazatel dotazem "koruna" na finanční, lesnické či panovnické dokumenty?
    - Synonyma
        - Vyhovuje dokument o "krychlích" dotazu na dokumenty o "kostkách"?
        - Vyhovuje dokument o "stromech" dotazu na "souvislé grafy bez cyklů"?
    - Hierarchie významů
        - Zvíře - Savec - Šelma - Medvěd
        - Tiskovina - Časopis
    - Ohebnost slov - Jít, Jde, Jdu, Jdou, ...
- Striktní booleovská logika není pro formulaci dotazů příliš vhodná
    - Dokument buďto vyhovuje nebo nevyhovuje
    - Dotazování v textech vyžaduje třídit odpovědi podle předpokládané vhodnosti pro tazatele - je potřebné mít možnost definovat míru shody dotazu s dokumentem

- Pozn. Možná dobré vědět něco o algoritmech prohledávání řetězců viz PT: Knuth-Morris-Pratt, Boyer-Moore, Brute Force, Rabin-Karp…

Předzpracování

- Databáze obvykle používají některý z booleovských modelů reprezentace dokumentů
    - nejlépe odpovídá běžným dotazům
    - relativně snadno se implementuje
    - dotazy jsou ve formě booleovských formulí, ve kterých operandy tvoří jednotlivá slova - řada různých modifikací
- 

Jednotlivé kroky předzpracování fulltextového vyhledávání:

Filtrace

- *Filtrace* - odstraní formátovací značky a nechá čistý ASCII text

Disambiguace

- *Desambiguace* - určí význam slova podle kontextu

- "pět chválu" ... sloveso pět
- "pět vozidel" ... číslovka pět

Lemmatizace

- *Lemmatizace* - určí základní tvar slova a gramatický tvar v dokumentu, často nahrazen pomocí stemmeru, který hledá kmen slova

Indexace

- *Indexace* - vytvoří pomocné seznamy lemmat a dokumentů a invertovaný soubor
  - dvojice *[id_dok, id_lemmatu]* setříděné dle id_lemmatu a zbavené duplicit
  - dnes obvykle více informací, např. pětice *[id_dok,č_odstavce,č_věty,č_slova,id_lemmatu]* - dovoluje vyhodnocování tzv. proximitních omezení na vzdálenost slov v dokumentu

Large OBjects (LOB)

- pro podporu vyhledávání je potřeba nad textovým sloupcem vytvořit index - invertovaný soubor
- běžné textové sloupce jsou pro tyto účely krátké a nevyhovující
  - obvykle se takto indexují sloupce některého z LOB (Large OBject) typů
- LOBy
  - standardní typy pro ukládání objemných dat na serveru, definováno v SQL-92 Full
  - až 4GB dat
    - BLOB - standardní binární typ
    - CLOB - znakový typ v univerzální znakové sadě
    - NCLOB - znakový typ v národní znakové sadě
  - v Oracle navíc externí typ BFILE
    - pouze pro čtení
    - samostatný binární soubor uložený vně databáe v OS
  - v MS SQL
    - Image - binární data do velikosti 2 GB
    - Text - textová data do velikosti 2 GB
    - NText - textová data v národní znakové sadě do vel. 1 GB
- Ve sloupcích tabulky je uložen pouze deskriptor (tzv. LOB lokátor), odkazující na samostatně uložená data

Dotazování

Oracle fulltext

- Filtrování vstupních dokumentů
  - **NULL_FILTER** - pro textové dokumenty TXT, HTML, XML
  - **INSO_FILTER** - pro binární dokumenty
  - **CHARSET_FILTER** - pro konverzi získaných dokumentů do znakové sady databáze
- Druhy fulltextových indexů
  - **CONTEXT**
    - základní typ indexu pro vyhledávání v textových datech
    - vhodný pro větší dokumenty
    - synchronizace indexu s daty je nutno provést explicitně
  - **CTXCAT**
    - vhodný pro menší dokumenty a jejich úryvky
    - může být zkombinován s dalšími netextovými sloupci pro kombinované dotazování
    - synchronizace indexu s daty se provádí automaticky se změnami v tabulce
  - **CTXRULE**
    - postaven na množině předdefinovaných dotazů
    - slouží pro klasifikaci dokumentů do skupin podle toho, kterým dotazům vyhovuje
- Uložení dokumentů
  - **NORMAL_DATASTORE** - text je v jednom sloupci jednoho řádků
  - **MULTI_COLUMN_DATASTORE** - text ve více sloupcích jednoho řádku
  - **URL_DATASTORE** - text je na internetu, dostupný přes URL ve sloupci

- Příklad vytvoření indexu nad textovým sloupcem:

```
CREATE INDEX myindex ON doc(htmlfile)
INDEXTYPE IS ctxsys.context
PARAMETERS('datastore ctxsys.default_datastore
filter ctxsys.null_filter
section group ctxsys.html_section_group');
```

- Spolu s novými typy indexů databáze implementují nové operátory pro porovnávání dotazu s textem
- Operátory vrací číslo - očekávanou míru shody obsahu textu s tazatelovými požadavky

Příklad dotazu:

```
strSql = "SELECT SCORE(1), file_name, filesize FROM my_doc " & _
    WHERE CONTAINS(content," & Search.Value & ", 1) > 0 " & _
    ORDER BY SCORE(1) DESC"
```

From <http://www.codeproject.com/Articles/12188/Full-text-search-with-Oracle-Text>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Možnosti tvorby datových skladů a metody dolování znalostí.

Thursday, May 30, 2013 8:22 AM

Základní problémy u běžných transakčních databázových systémů:

- nedosažitelnost dat skrytých v transakčních systémech
- dlouhá odezva při plnění komplikovaných dotazů
- složitá, uživatelsky nepříjemná rozhraní k databázovému softwaru
- cena v administrativě a složitost v podpoře vzdálených uživatelů
- soutěžení o počítačové zdroje mezi transakčními systémy a systémy podporujícími rozhodování

Cesta k řešení těchto problémů = datové sklady, tzv. Data Warehouse – DW

Datawarehouse

• DW označují db architekturu používanou pro údržbu historických dat, která jsou získána z jedné nebo více operativních db. Typicky, tato data jsou vyčištěna a restrukturována pro podporu dotazů, agregací a analýz.

• Klíčové: integrace vlastních + externích dat

Modely & Operátory warehousu

• Datové Modely
– relační
– hvězdice & vločky
– krychle
• Operátory
– slice & dice (řez & výřez)
– roll-up, drill-down (srolování, zavrtání)
– pivoting
– další

Komponenty DW

• **akvizice dat a jejich integrace** do DW (generátory kódu, replikace dat, middleware, kopírování)

• **řízení dat** (databázový server + služby: archivace, autorizace, zálohování a zotavení z chyb, provoz, monitorování a ladění, řízení zdrojů)

• **slovník informací** (metadata a přístup k nim)

• **přístup k datům** a komponenty dodání dat (db middleware, OLAP, multidimenzionální data, data řízená časem a událostmi)

Velká diskuze: E-R vs. multidimenzionální přístupy

2 přístupy k Datovému Modelování:

· **konceptuální struktury založené na tabulkách** (dimenzionální a tabulky faktů) organizovaných do tzv. hvězdicových schémat,

· **konceptuální struktury jsou založeny na hyperkostkách** (kostkách, multidimenzionálních polích), které reprezentují data jako multidimenzionální strukturu.

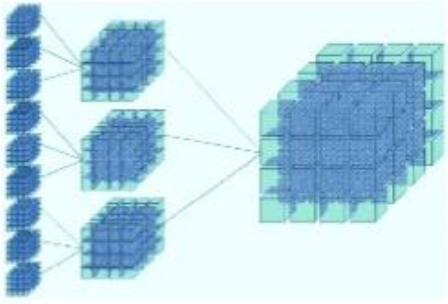
- samostatný informační systém postaven na již pořízených datech, určen především k jejich analýze
- architektura založená na ~~relačním~~ SŘBD, která se používá pro údržbu historických dat získaných z databází operativních dat, jenž byla sjednocena a zkontrolována před jejich použitím v databázi DW
- data z DW jsou aktualizována v delších časových intervalech, jsou vyjádřena v jednoduchých uživatelských pojmech a jsou sumarizována pro rychlou analýzu

- DW je obrovská databáze obsahující data za dlouhé časové období
- často slučuje data z více rozdílných zdrojů, které mohou obsahovat data různé kvality nebo používat nejednotné formáty a reprezentace
- objemově zabírá stovky GB až několik TB
- nemusí být databází v běžném smyslu, tj. pro přesné provádění transakcí
- je určen pro rychlé vyhledávání
- nejsou kladeny nijak důrazné požadavky na správnost a úplnost dat

Charakteristika

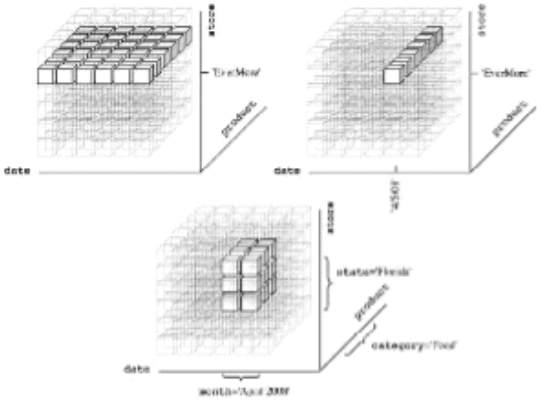
- data jsou uložena na různých místech ve formě relačních tabulek
  - uživatelé mohou tabulky jen číst
  - zapisovat může aktualizační program pravidelně udržující tabulky
- dotazy jsou většinou komplexní
  - podporují tzv. on-line analytické zpracování (OLAP)
    - výrazně se liší od on-line transakčního zpracování (OLTP)
    - operační databáze je přizpůsobena pro podporu OLTP
      - složité OLAP dotazy by vyústily do nepřijatelné odezvy
    - typické OLAP operace
      - **roll-up** (zvýšení stupně agregace)

Roll-up



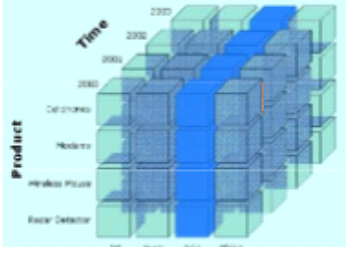
      - **drill-down** (snížení stupně agregace)

Drill down

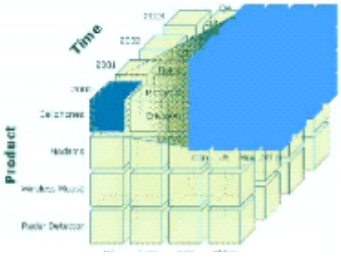


      - **slice-and-dice** (selekce a projekce)

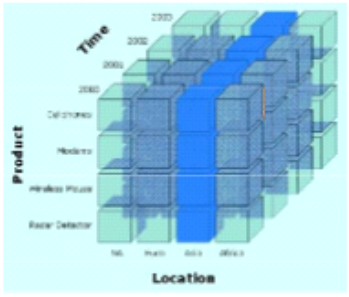
Slice



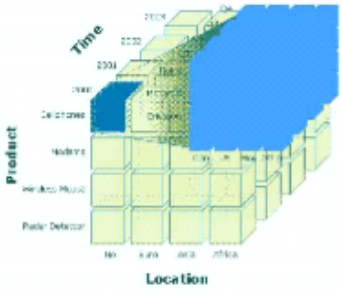
Dice



Slice

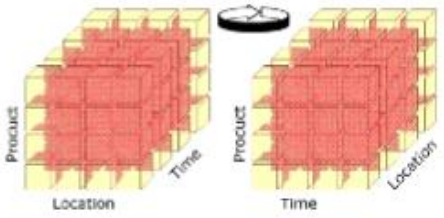


Dice



- **pivot** (přeorientování vícerozměrného pohledu na data)

Pivot



- na základě dotazu se pospojují potřebná data do vícerozměrné tabulky (nebo více tabulek), do kterých lze klást SQL dotazy
- pro častější dotazy si uchovávají předem připravené vícerozměrné tabulky
- zátěž je většinou způsobena složitými dotazy, jež přistupují k miliónům záznamů a provádějí množství operací
- data bývají modelována vícerozměrně
  - v obchodním data warehouse mohou těmito rozměry být např. čas prodeje, místo prodeje, prodavač, výrobek, …
  - rozměry mohou být i hierarchické např. čas prodeje jako den-měsíc-čtvrtletí-rok, zboží jako výrobek-kategorie-průmysl
  - spojení více tabulek pomocí odkazu na řádky jednotlivých tabulek
  - používají speciální organizaci dat, přístupové a implementační metody, jež obecně nejsou v komerčních databázových systémech určených pro OLTP podporovány

Databázový systém – OLTP (Online Transaction Processing Systems)

- zákaznicky orientovaný
- aktuální data -- lze považovat i za slabinu, při výpadku (chybě), vznikají ztráty pro byznys
- ER schéma
- sofistikované atomické transakce i přes několik systémů(bank, po síti,...)
- velikost DB až několik GB
- jednoduché a efektivní
- příkladem je bankomat

DataWarehouse – OLAP (Online analytical Processing)

- orientovaný na trh, rychlé (oproti OLTP) získání výsledků na analytické dotazy
- historická data, multidimenzionální datový model
- agregovaná data (nenormalizovaná=redundantní)
- schéma hvězdy či vločky
- převážně pouze čtení
- velikost až TB
- použití: byznys reporty o prodeji, marketing, management reporty, rozpočty, finanční předpovědi a reporty

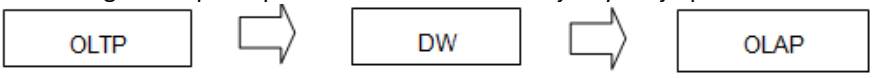
OLAP (**Online Analytical Processing**) je technologie uložení dat v databázi, která umožňuje uspořádat velké objemy dat tak, aby byla data přístupná a srozumitelná uživatelům zabývajícím se analýzou obchodních trendů a výsledků (Business Intelligence). Způsob uložení dat se svým zaměřením liší od běžněji užívaného **OLTP** (**Online Transaction Processing**), kde je důraz kladen především na snadné a bezpečné ukládání změn v datech v konkurenčním (víceuživatelském) prostředí.

Vloženo z <http://cs.wikipedia.org/wiki/OLAP>

Použití DW

- prezentace dat
- testování hypotéz
- objevování nových informací

Architektura DataWarehouse

- tři úrovně:
- klient
- OLAP server (MOLAP/ROLAP server)
- databázový server DW
- data lze organizovat v tzv. multidimenzionálním datovém modelu
  - odlišný od modelu relačního
  - odpovídá mu specializovaný software, multidimenzionální SŘBD (MDD)
  - model připomíná techniku spreadsheet ve více než dvou rozměrech
  - data jsou implementována pomocí vícerozměrných polí, jejichž dimenze odpovídají dimenzím podnikání organizace
- navržení a vytvoření DW je proces skládající se z následujících bodů:
  - definovat architekturu, umístění a rozčlenění dat a fyzickou organizaci
  - naplánovat kapacitu, vybrat OLAP servery a nástroje
  - spojit servery, klientské nástroje, zdroje přes gatewaye, drivery ODBC, ...
  - navrhnout schéma a pohledy, přístupové metody, některé složité dotazy
  - mít skripty pro získávání, čištění, transformaci, ukládání a aktualizaci dat
  - vytvořit koncové uživatelské aplikace
  - spustit data warehouse i aplikace
- vytvoření je složitý proces trvající mnohdy i několik let
- mnoho organizací proto používá Data Mart umožňující rychlejší práci
- 

Datová tržiště (Data Mart)

- DW slouží jako základna pro extrakci množin dat, resp. jejich agregaci do dílčích (replikovaných) MDD (Multidimenzionální DB)
  - MDD může pro DW sloužit ve dvou rolích
    - ”front-end” pro DW a poskytovat uživateli služby pro realizaci analytického zpracování (DW/OLAP)
    - “front-end” jednomu (několika) systémům OLTP - alternativa za DW, tj. poskytnout uživateli s OLTP data analytickým způsobem (OLTP/OLAP) – jde vlastně o datové tržiště

Systém OLAP (OnLine Analytical Processing)

- na databázové stroje jsou kladeny specifické požadavky
- objem zpracovávaných dat
- transakční systém o velikosti gigabajtů dosáhne použitím jen jedné dimense velikosti desítek či stovek gigabajtů
- rychlost odezvy analytického systému je důležitá
- počet uživatelů současně pracujících s databází není zajímavý
  - počet pracovníků vyššího managementu je omezen
  - pro pracovníky nižších stupňů bývají údaje z datových skladů převedeny do menších specializovaných databází – datových tržišť
- s těmito omezeními se vyrovnává dvojím způsobem
- uzpůsobení stávajících systémů pro práci s vícerozměrovými daty
- přidáním modulu, který to zajišťuje a prostředků pro jeho ovládání
- v lepším případě mění způsob uložení dat, v horším“překládá” operace s

vícedimenzionálními daty na operace s daty relačními
- vytvoření speciálního systému správy dat, určeného pouze pro OLAP
- umožňuje provést maximum optimalizací vzhledem k nárokům, jež jsou kladené analytickým způsobem práce - převažující způsob

Programy pro vytváření a plnění databáze (ETL - viz SI)

- převodní programy
  - načtení data z několika databází, či souborů a udělat z nich novou databázi, agregace se musí naprogramovat
- systémy znázorňující převodu dat graficky a administrátor dat namapuje zdrojová data do struktur vytvářeného datového skladu
  - výsledkem jsou buď programy (scripty) nebo přímo vykonání funkce
- moduly pro plánování jednotlivých akcí

Nástroje pro práci s daty - poslední trendy v architektuře klient/server

- nabízejí variantu tenkého klienta v podobě HTML prohlížeče

Reporting, monitorování, ad-hoc dotazy

- programy umožňující kladení dotazů a formátování odpovědí
  - nejčastěji jde o vizuální dotazovací nástroje
  - makra v tabulkovém procesoru
  - uživatelské rozhraní různě propracované:
    - zadání seskupení výsledku podle různých kritérií
    - formální kontrola dotazů
    - vytváření slovníků a metadat

MOLAP - Multidimenzionální OLAP

- datová krychle (obsahuje fakta)
- hierarchické dimenze (částečné či totální uspořádání)
  - **vločkové schéma** -- hlavní tabulka faktů je v relaci s dimezionálními tabulkami, přes cizí klíče, dimenzionální tabulky mohou být také v relaci s dalšími subdimenzionálními tabulkami podobně jako hlavní tabulka faktů; vytváří hierarchie dimenzí
  - **hvězdové schéma** -- je speciální případ vločkového, dimenzionální tabulky již nejsou v relaci s dalšími subdimenzionálními tabulkami; žádné hierarchie, jednodušší

ROLAP – Relační OLAP

- na relační architektuře založený model DW strukturou propojených DB tabulek - Relační OLAP (ROLAP) – pomalejší zpracování než MOLAP
- užívá relační nebo rozšířený relační DBMS, např server METACUBE Informix, pracuje s relačními tabulkami uspořádanými do hvězdy/vločky, adresuje pomocí klíče, data jsou neagregovaná)

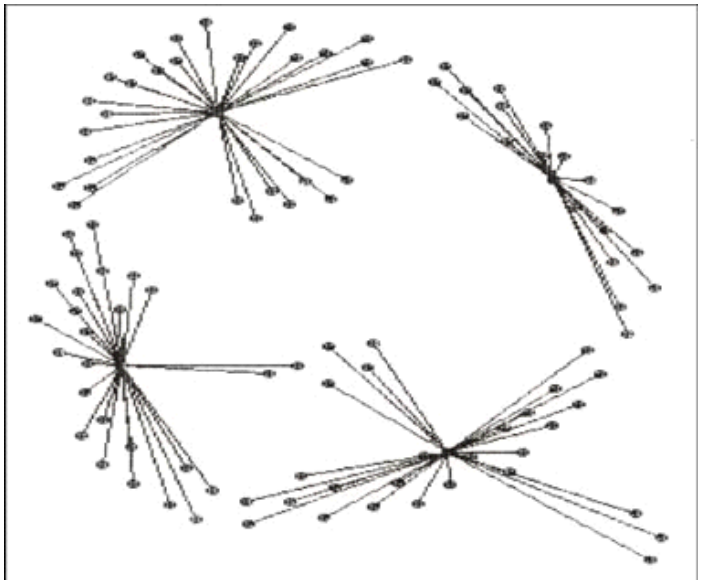
Metody dolování znalostí

Asociace

- Klasické – mezi dvěma podmnožinami atributů
- Transakční – v rámci množiny atributů
- Agregované – mezi podmnožinou atributů a jejich charakteristikami

Algoritmy generující asociace:
- Triviální
- Uspořádané generování pravidel
- Vzorkování
- K-množiny
- …

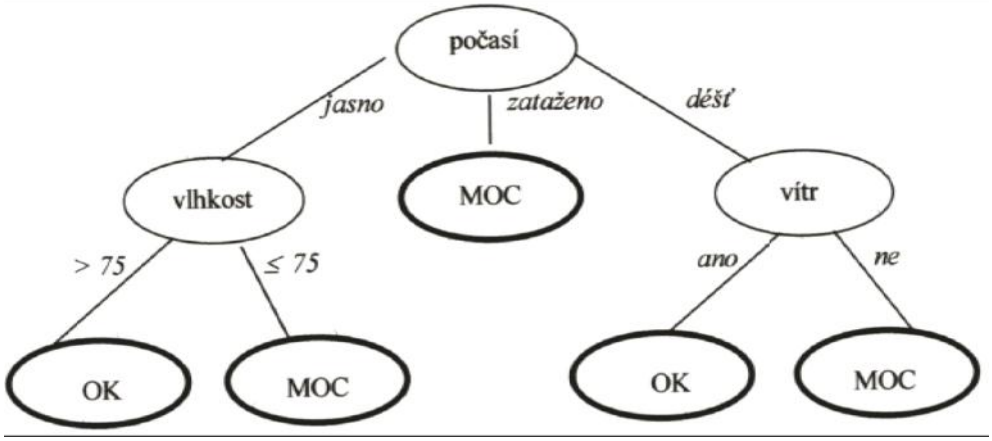
Shlukování



-analýzuje, zda se množina objektů přirozeně rozpadá na výrazné podmnožiny (shluky) objektů vzájemně si podobných a přitom nepodobných objektům podmnožin ostatních
- případně dále analyzuje, zda existuje celá hierarchie takových rozkladů
- pokud shluky existují, čím jsou charakteristické
- jak se případné další objekty zařadí do již definovaných shluků
Shluková analýza netvoří ucelenou teorii, ale je to řada metod založených na různých principech (různorodost řešených problémů, požadovaných typů výsledků, velká data, neurčitost definice shluku)

Metody dle cíle shlukování
- **hierarchické –** produkující hierarchii rozkladů, kde každý rozklad je zjemněním předcházejícího
- **nehierarchické –** produkující prostý rozklad objektů na podmnožiny
Metody dle typu výsledných shluků
- shluky kulové, body soustředěny kolem svého těžiště
- shluky obecné tvoří souvislé husté oblasti nejrůznějších tvarů
Metody dle typu rozkladu
- shluky disjunktní
- shluky překrývající se
Algoritmy
- **nehierarchické** (optimalizační k-středové, analýzy módů, fuzzy k-středové, neuronové sítě)
- **hierarchické** (aglomerativní, divizivní)
- **vzorkování**

Rozhodovací stromy



Poznámky :
metoda k-středové = k-means

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Metody dolování (jiný zdroj, ne z přednášek)

1. popis dat (asi asociace?)
   - najít charakteristiky dat
   - často ve spojení s dalšími metodami, např. segmentací (hledám charakteristiky segmentů)
2. shlukování (clustering)
   - hledání a vytváření kategorií, do kterých jsou data uspořádána
   - nemusí být disjunktní, jeden objekt klidně ve více skupinách
   - tím se shlukování liší od klasifikace!!
3. klasifikace
   - rozdělování objektu do jednotlivých tříd podle cílového atributu (nespojitá veličina)
   - předpokládá se, že jednotlivé třídy, do kterých objekty rozděluji, jsou předem známé
4. regresní analýza
   - obdoba klasifikace; cílový atribut je spojitá veličina
5. analýza závislosti
   - hledám takový model, který popisuje vztahy mezi daty
   - např. analýza spotřebního koše (co se s čím často prodává)
   - hrozí, že naleznu neexistující závislost

Zavádění datového skladu

strategie velkého třesku (hub architecture)

- buduji celý sklad naráz
- vytvoření celopodnikového datového modelu
- datová tržiště modelována dimenzionálně a napojena na samotný DW

strategie postupného budování datových tržišť (bus architecture)

- z nich pak dohromady vznikne datový sklad
- výhodou je jednoduchost a srozumitelnost datového modelu, iterativnost procesu

Dva způsoby budování datového skladu

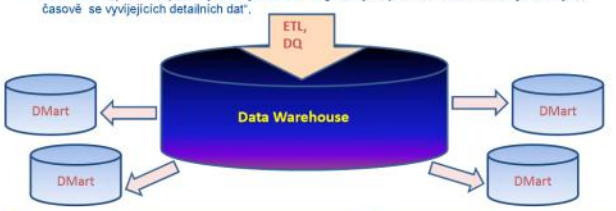
1. **Data warehouse jako množina data martů (bus architecture)**
   Ralph Kimball: "Data warehouse není nic jiného než sjednocení data martů,.

Data marty je možné sjednotit pouze za předpokladu tzv. "všeobecně přijatých" dimenzí a faktů (conformed dimension). V opačném případě není možné DM spojovat do 1 celku resp. pokud by se spojovaly, výsledkem budou špatná data!

Plus	Minus
Rychlá implementace data martů.	Redundance dat.
Nízké počáteční náklady.	Každý DM má vlastní historii, ETL, dimenze, řešení datové kvality.
	Hůř monitorovatelné procesy, vyšší HW i SW nároky na údržbu.

Dva způsoby budování datového skladu

2. **Centrální Data Warehouse (hub architecture)**
   Bill Inmon: „(Centrální) datový sklad je soubor integrovaných, předmětově orientovaných, stálých, časově se vyvíjejících detailních dat".

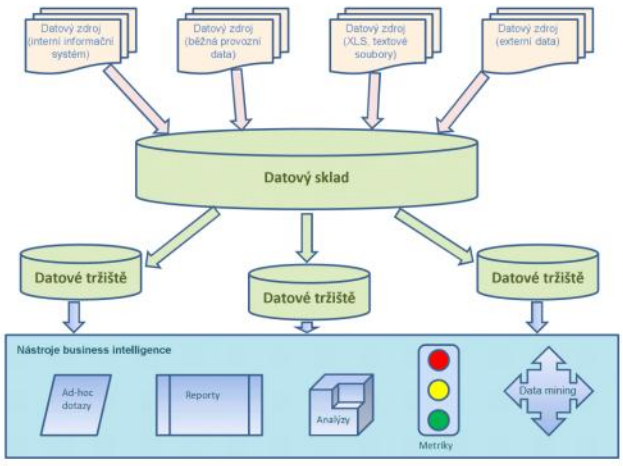


Plus	Minus
Centrální datový sklad plněný jednotným ETL postupem, použito 1 řešení datové kvality a vytvořeny společné dimenze.	Vyšší náklady na návrh a implementaci centrálního skladu.
Minimalizována redundantnost dat.	Delší „přípravný" čas.
Možnost centrálního monitorování datového skladu	

ODS je soubor integrovaných, předmětově orientovaných, nestálých aktuálních detailních dat vytvořená pro aktuální potřeby uživatelů.

Porovnání transakčních systémů (OLTP) a analytických systémů (OLAP)

Znak	OLTP	OLAP
Charakteristika	Provozní zpracování	Informační zpracování
Orientace	Transakční	Analytická
Uživatel	Běžný uživatel, databázový administrátor	Znalostní pracovník (manažer, analytik)
Funkce	Každodenní operace	Dlouhodobé informační požadavky, podpora rozhodování
Návrh databáze	Entitně-relační základ, aplikačně orientovaný	Hvězda/sněžná vločka, věcná orientace
Data	Současná, zaručeně aktuální	Historická
Sumarizace dat	Základní, vysoká podrobnost dat	Shrnutá, kompaktní
Náhled	Detailní	Shrnutý, multidimenzionální
Jednotky práce	Krátké, jednoduché transakce	Komplexní dotazy
Přístup	Číst, pořizovat a aktualizovat	Pouze číst
Zaměření	Vkládání dat	Získávání informací
Počet dostupných záznamů	Desítky	Miliony
Počet uživatelů	Stovky – tisíce.	Desítky – stovky
Velikost databáze	100 MB až GB	100 GB až TB
Přednosti	Vysoký výkon, vysoká přístupnost	Vysoká flexibilita, nezávislost koncového uživatele
Míry hodnocení	Propustnost transakcí	Propustnost dotazů a doba odezvy



Informační systémy, jejich základní vlastnosti a typy.

Wednesday, May 29, 2013 4:49 PM

Informační systém (IS) je systém pro sběr, udržování, zpracování a poskytování informací.

Funkce informačního systému

Konkrétní procesy (činnosti) podporující základní cíle informačního systému:

- získávání informací
- zpracování informací (evidence, organizace – pořádání, kategorizace, konverze –změna média, třídění, vyhledávání, agregace, odvozování nových informací, dolování znalostí)
- uložení informací (zaznamenávání a archivace dat, datová úložiště a datové sklady)
- přenos informací (v rámci počítačových sítí)
- zpřístupnění informací (tisk, zobrazení, vizualizace, šíření…)

Následující body vystihují vlastnosti, které by kvalitní IS s maximální výkonností měl splňovat.

- Musí obsahovat nutné informace, které uchovává, analyzuje a s potřebnou rychlostí předává procesům. Dané informace se týkají zejména vlastní činnosti firmy jako je výroba, evidence zákazníků, zásob, zaměstnanců, finance, stav a vývoj vlastních výrobků.
- Musí obsahovat informace o konkurenci, světovém trhu, trendech výroby, optimalizaci výrobních procesů, o místech působnosti firmy, o strategických cílech a podobně.
- Musí obsahovat moduly pro zjednodušení a urychlení výroby, čímž je míněno hlavně urychlení a zefektivnění návrhu výrobků, technologická příprava výroby a její řízení.
- Musí umožňovat rychlou komunikaci pracovníků firmy, jednotlivých pracovních úseků, ale musí také zahrnovat komunikaci se světem.
- Musí umožňovat z dostupných informací zpracovávat cíle a strategie firmy, koordinovat činnost různých procesů a tím přispívat k zefektivnění činnosti firmy.
- Musí nabízet rychlou komunikaci se zákazníkem přes počítačovou síť.
- Musí obsahovat další nutné moduly k vedení firmy jako jsou statistiky, mzdy, účetnictví, kompletní personalistika, sklad, oblast manažer – marketing, výroba a další.

Typy informačních systémů

Podnikové informační systémy (BIS – business information system)

Systémy, provozované v kontextu konkrétní organizace, jejichž účelem je správa informací a znalostí a jejich integrace do podnikových procesů.
Obsažené informace jsou chápány jako jeden z ekonomických zdrojů (aktiv) organizace. Rozlišují se systémy podporující:

- vlastní činnosti a služby organizace (automatizace podnikových procesů – např. CIM, workflow management, elektronický obchod, systémy pro tvorbu a správu dokumentů)
- manažerské systémy, podporující řídící a administrativní funkce. Jako softwarové vybavení se nabízejí zpravidla tzv. typová řešení pro konkrétní odvětví nebo obchodní model.

Provozní, transakční systémy

- **ERP** – enterprise resources planning: systémy na podporu provozu (chodu) firmy
  Technologický princip: aplikační software, OLTP (Online Transaction Processing), relační databáze. OLTP je technologie uložení dat v databázi, která umožňuje jejich co nejsnadnější a nejbezpečnější modifikaci ve více uživatelském prostředí. Jedná se o přístup používaný v současné době v převážné většině databázových aplikací.

Systémy na podporu plánování

- **SCM** – supply chain management:plánování dodavatelských logistických řetězců
- HR – human resources – řízení lidských zdrojů
- APS – advanced planning and scheduling: systémy na podporu vnitropodnikového (dílenského) plánování,

Systémy řízení vztahů se zákazníky

- **CRM** – customer relationship management: Shromažďování, zpracování a využití informací o

zákaznících firmy za účelem poznat, pochopit a předvídat potřeby, přání a nákupní zvyklosti zákazníků. Podporuje komunikaci mezi firmou a jejími zákazníky.

Systémy na podporu rozhodování

- **BI** – business intelligence)
  Technologický princip: OLAP (Online Analytical Processing), datové sklady (data warehouse), dolování dat (data mining) – základem není realizace transakcí, ale prohledávání a analýza velkých objemů dat
- MIS – management information system,
- EIS – executive information system,
- DSS – decision support system,

Systémy pro tvorbu a správu dokumentů

- **DMS** – document management system
  Systémy umožňující efektivní práci s elektronickými dokumenty a jejich obsahem v průběhu celého jejich životního cyklu.
  Typickými procesy jsou tvorba, schvalování, evidence, digitalizace, prohlížení, editace, publikování, komunikace, sdílení, uložení, vyhledání, archivace, skartace apod.

  Obvykle je zahrnuta i skupinová spolupráce, workflow management a propojení dokumentů s informacemi v ostatních (např. provozních) informačních systémech.

  Technologický princip: aplikační software, obsahující nástroje pro tvorbu, publikování, fulltextové vyhledávání, řízení přístupu k elektronickým dokumentům, správu verzí, sledování historie použití a změn.

- DTP – desktop publishing

Knihovní systémy

Systémy určené k automatizaci procesů realizovaných v knihovně. Obvykle mají modulární strukturu; typické moduly jsou akvizice, katalogizace, výpůjčky apod. Zpravidla obsahuje i nástroje pro zapojení do sítě knihoven a pro komunikaci s externími zdroji.
Technologický princip: aplikační software provozního (transakčního) typu,

Geografické informační systémy (GIS)

Prostorově orientované informační systémy, provozované za podpory informačních a komunikačních technologií. Datovou základnu tvoří digitální geografické informace ve formě záznamů nebo objektů (tzv. geoprvky), s nimiž specializovaný software umožňuje provádět manipulaci (zápis a editace údajů, uložení, vyhledávání, propojování, transformace a vizualizace), lokalizaci (určení polohy), geografické analýzy a modelování (např. trojrozměrný model terénu).

Expertní systémy

Počítačové aplikace nebo systémy simulující poznávací a rozhodovací činnost experta při řešení složitých úloh s cílem dosáhnout ve zvolené problémové oblasti kvality rozhodování na úrovni experta.
Technologický princip: základní součásti tvoří báze znalostí, báze dat (faktů) k řešeným případům a řídící mechanismus (inferenční neboli odvozovací stroj, rozhodovací jádro), tj. program pro práci s těmito bázemi využívající technik umělé inteligence. Tyto základní součásti obvykle doplňuje modul pro komunikaci s uživatelem

Další členění informačních systémů

Veřejné informační systémy

Informační systémy, které jsou dostupné široké veřejnosti a poskytují veřejné informační služby. V tomto smyslu se jedná o jakékoli informační systémy bez ohledu na jejich provozovatele, obsah, typ, formu a příp. cenu poskytovaných informací a služeb. Opakem jsou tzv. privátní, uzavřené, neveřejné informační systémy (např. podnikové informační systémy, systémy zajišťující obranu státu, osobní informační systémy ad.).

Státní informační systém, informační systém veřejné správy

Systém, jehož účelem je podporovat činnosti provozované při výkonu veřejné správy, tj. státní správy a samosprávy, a poskytovat veřejné informační služby včetně informací o subjektech veřejné správy. Představuje

komplex navzájem propojených subsystémů, členěných z hlediska věcného, resortního a regionálního. Nejdůležitější součást datové základny tvoří evidence (registry) základních skutečností nezbytných pro výkon veřejné správy: evidence obyvatel, evidence ekonomických subjektů, evidence území a územních jednotek.

eGovernment

Moderního, přátelského a efektivního úřadu

eHealth

Elektronické zdravotnictví (eHealth) je souhrnný název pro řadu nástrojů založených na informačních a komunikačních technologiích, které podporují a zlepšují prevenci, diagnostiku, léčbu, sledování a řízení zdraví a životního stylu.

eLibrary

Computer aided technologie (CAD, CAM, CIM, CASE...)
Podstata: počítačová podpora (automatizace) některých procesů (návrh, výroba ap.)
CAD (computer-aided design) počítačem podporované projektování. Jde o velkou oblast IT, která zastřešuje širokou činnost navrhování.
Ve strojírenství CAM (computer-aided manufacturing) CAE (computer-aided engineering)

Stavebnictví a architektura – AEC (Architecture-Engineering-Construction), BIM (Buliding Information Model), CAAD (Computer-aided architectural design)

Rozdělení dle technologie zpracování dat

- OLTP - umožňují skupině uživatelů vykonávat bezprostředně (online) velké množství transakcí
  - relační databáze
- OLAP - analýza velkého množství údajů, většinou jen pro čteni, nadstavba OLTP
  - např. BI

Rozdělení IS dle uživatelů

- Veřejné informační systémy - Informační systémy, které jsou dostupné široké veřejnosti a poskytují veřejné informační služby. V tomto smyslu se jedná o jakékoli informační systémy bez ohledu na jejich provozovatele, obsah, typ, formu a příp. cenu poskytovaných informací a služeb.
- Privátní, uzavřené, neveřejné informační systémy (např. podnikové informační systémy, systémy zajišťující obranu státu, osobní informační systémy ad.).

Analýza informačních systémů (IS), role modelování a metodik při tvorbě IS.

Thursday, May 30, 2013 8:23 AM

Informační systém

(IS) je systém pro sběr, udržování, zpracování a poskytování informací.

Informační systémy jsou založené na informačních a komunikačních technologiích. V poslední době se používá zkratka ICT (Information and Communication Technologies).

IS obsahuje data, znalosti a informace

- *Data* -jakékoli vyjádření (reprezentace) skutečnosti, schopné přenosu, uchování, interpretace či zpracování. Umožňují přenášet a zpracovávat odraz skutečnosti.
- *Znalosti* - výsledek poznávacího procesu, předpoklad uvědomělé činnosti. To, co jednotlivec ví po osvojení dat a po jejich začlenění do souvislostí. Účel znalostí - porozumět realitě.
- *Informace* - je definovaná pomocí dat a znalostí - data, která mají smysl (význam), sdělitelné (komunikovatelné) znalosti.
-

Role modelování a metodiky pří tvorbě IS

Složitost systému se promítá do složitosti jeho návrhu a realizace. (tedy i do modelování)

Role modelování a metodiky je taková, že odstraňuje tyto problémy (asi)

- Systém dělá něco jiného než by měl
- Systém řeší problémy lokálně. Důsledkem je, že jedna a tatáž věc je řešena na různých místech několikrát, po každé jinak.
- Opravy a změny systému jsou velmi obtížné a drahé. (Jestliže opravujeme chybu na základě lokálních znalostí, tak vlastně opravujeme výskyt chyby, ale ne její příčinu.)
- Systém nelze realizovat několika skupinami současně, paralelně. Bez plánu vznikají komunikační problémy.

Metodiky

Chceme-li se vyhnout potížím s lokálním rozhodováním, musíme postupovat **metodicky (ne chaoticky)**, strukturálně, dle „dobrých“ osvědčených vzorů.

Návod jak postupovat nám dávají **ověřené postupy** – vypracované metodiky.

Metodiky odrážejí určité náhledy na „realitu“, říkají **„jaké“** kroky učinit v jakém pořadí a **„jak“** je provádět. Dobré metodiky nám říkají i **„proč“** to tak má být.

Metodiky jsou **konservovanou zkušeností** několika generací programátorů a projektantů. Zobecnění principů, zásad, které se osvědčily, viz historie UML.

Modely

Místo abychom se snažili popsat systém jako celek, vytváříme na něj **jednotlivé pohledy** – jeho jednotlivé, dílčí modely. Díváme se na systém postupně z jednotlivých „míst pozorování“, z jednotlivých perspektiv.

Díváme-li se na systém z jednoho místa, opomíjíme vlastnosti z tohoto místa „neviditelné“, nepodstatné a tím si práci zjednodušíme tak, že je mentálně zvládnutelná. Jednotlivé pohledy jsou jednodušší, zvládnutelné. Opomíjené vlastnosti se neztratí, jsou hlavními vlastnostmi v jiných pohledech - modelech.

Pohledy musíme volit tak, že postupně popíšeme všechny relevantní vlastnosti systému. Postupně popíšeme vše, co potřebujeme k dosažení stanoveného cíle.

Z jednotlivých pohledů lze zpětně zrekonstruovat celý systém (počítačová tomografie).

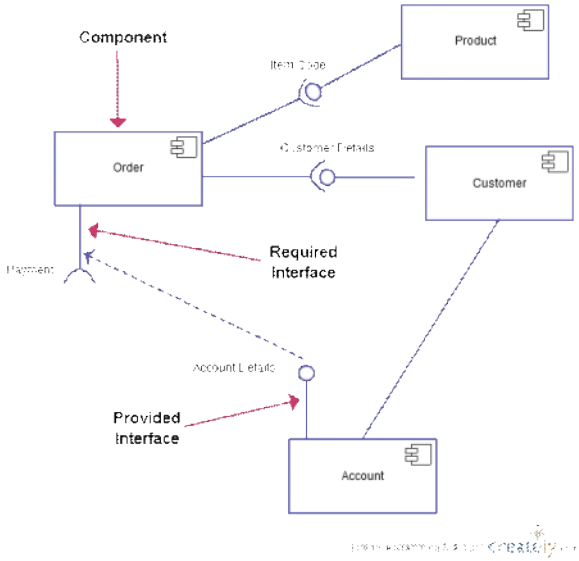
Pro tvorbu různých pohledů jsou obvykle **využity diagramy** – grafické objekty, jejichž kombinací lze tyto pohledy vytvářet.

Diagram je **graficky znázorněný model**. Diagram popisuje jistou část modelu pomocí grafických symbolů.

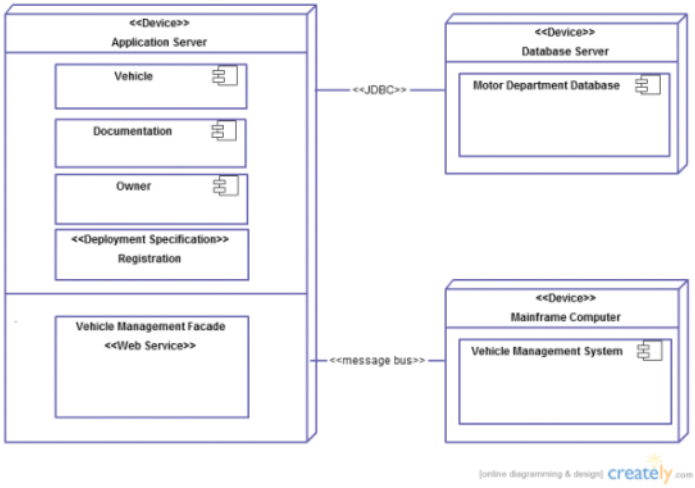
Tento přístup lze přirovnat k modelu stavby, který je tvořen syntézou dílčích stavebních plánů odpovídajících specifickým pohledům na stavbu – plán hrubé stavby, plánu rozvodů elektřiny, plánu rozvodů vody, … V každém z těchto plánů jsou zobrazeny pouze elementy modelu podstatné pro daný pohled, od ostatních elementů modelu je abstrahováno. **Pohledy nejsou nezávislé**, dohromady tvoří konzistentní pohled na systém, tedy konzistentní model.

Pro tvorbu diagramů systému, jejichž syntézou bude model, definuje např. **UML devět typů diagramů.**

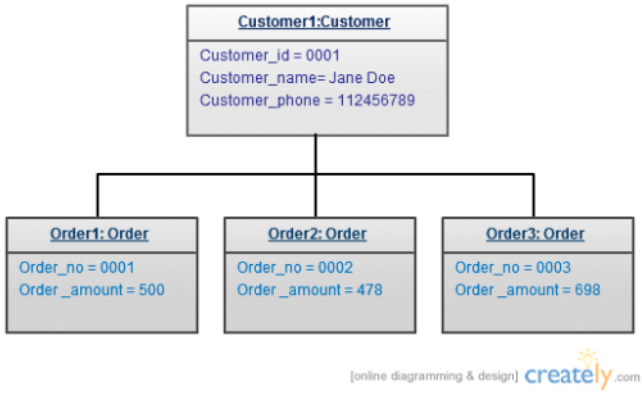
1. Class Diagram
2. Component Diagram



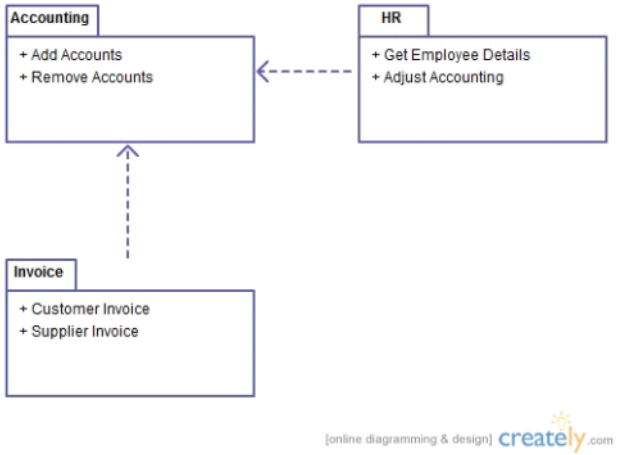
3. Deployment Diagram



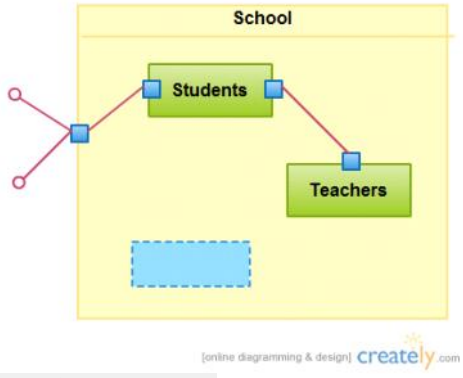
4. Object Diagram (Instance diagram)



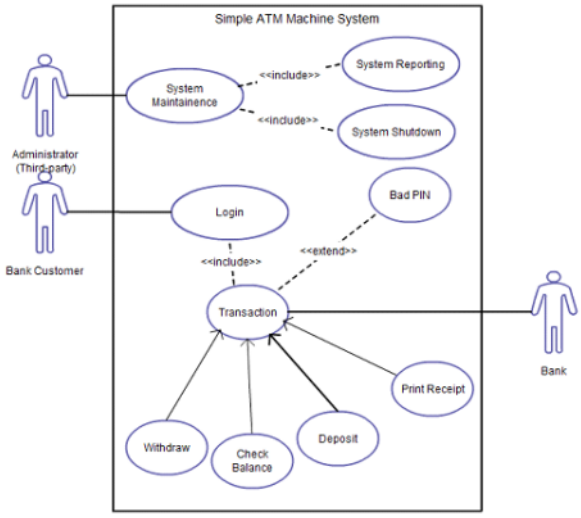
5. Package Diagram



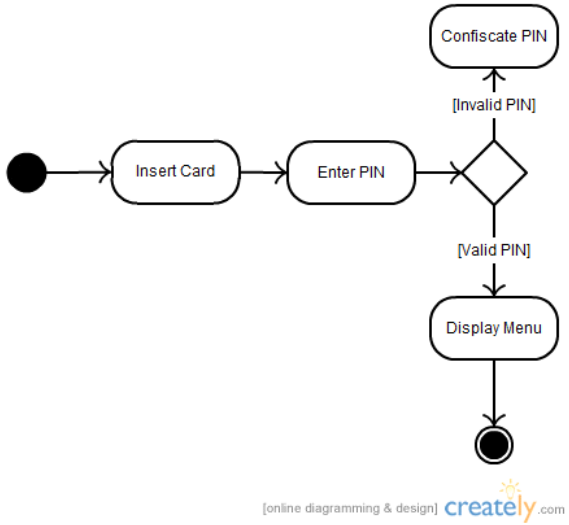
6. Profile Diagram (? Asi ignorovat, od UML 2)
7. Composite Structure Diagram (popis vnitřní struktury třídy)



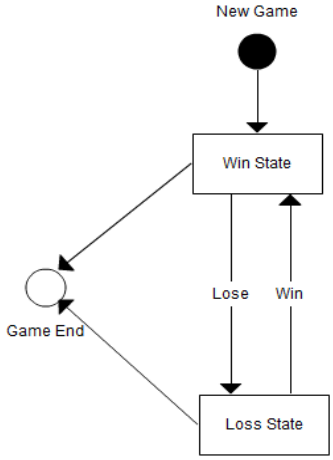
8. Use Case Diagram



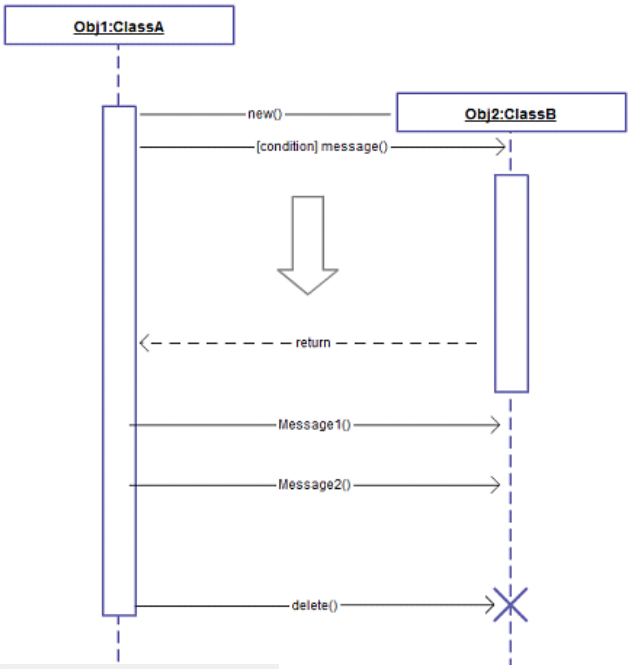
9. Activity Diagram



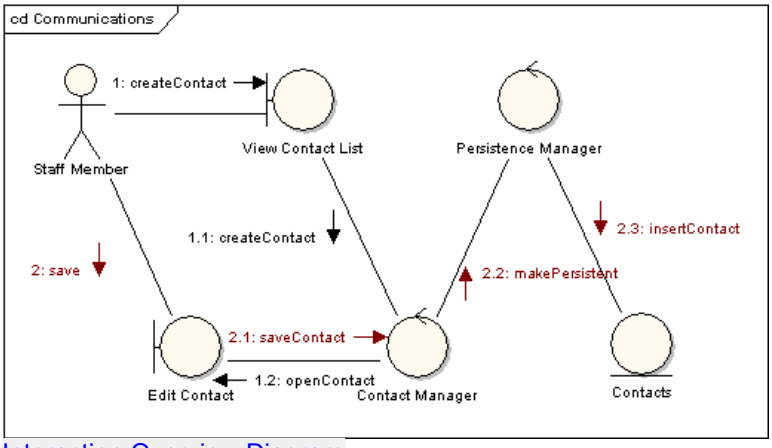
10. State Machine Diagram



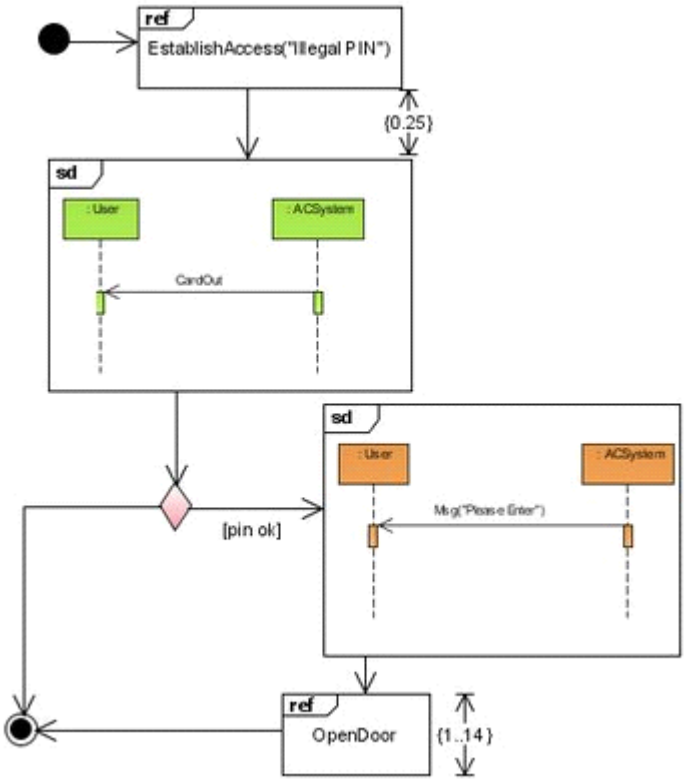
11. Sequence Diagram



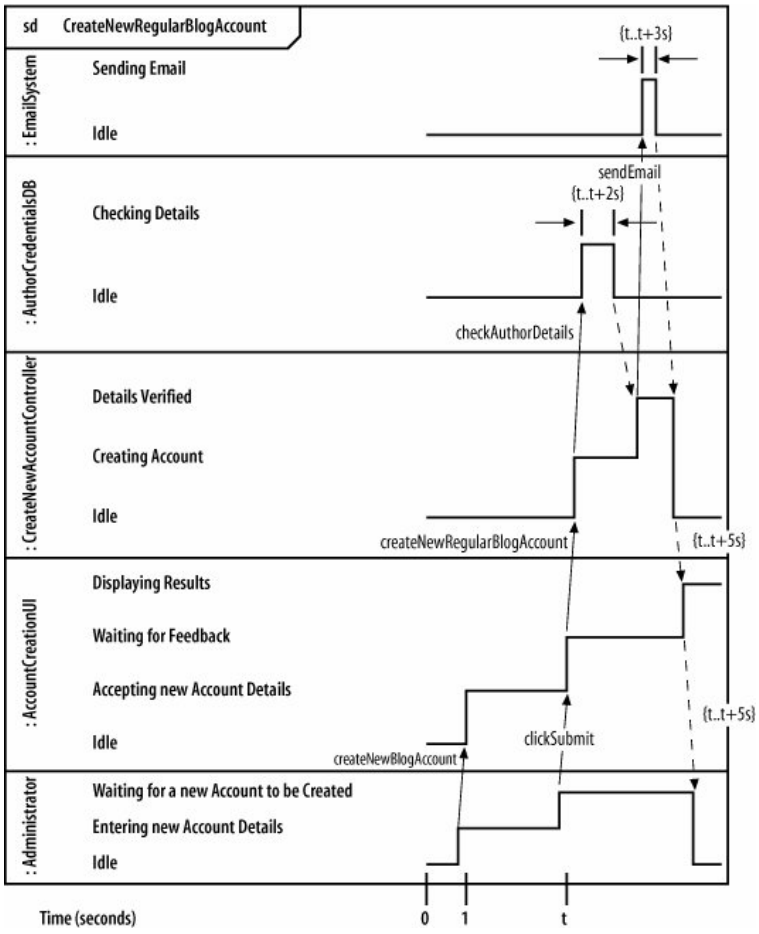
12. Communication Diagram



13. Interaction Overview Diagram



14. Timing Diagram (asi ignore)



Z <http://creately.com/blog/diagrams/uml-diagram-types-examples/>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Metodika návrhu a realizace informačního systému – strukturální a objektová analýza.

Thursday, May 30, 2013 8:23 AM

Existuje v zásadě několik metodik, které popisují analýzu, návrh a realizaci informačních systémů

Jedná se o více méně podrobný popis postupu, který vede návrháře jasně definovanými fázemi krok po kroku při vytváření IS

Metodiky jsou často obecné a každá firma si je může přizpůsobit pro vlastní potřebu a prostředí

Mezi nejpoužívanější patří Strukturální analýza (ta je nejstarší), SSADM (vznik 80. léta ve Velké Británii) a Objektová analýza (konec 80. let, 90. léta)

Metodiky návrhu a realizace informačního systému

Fenomén úhlu pohledu
Chceme-li se vyhnout potížím s lokálním rozhodováním, musíme postupovat metodicky (ne chaoticky), strukturálně, dle „dobrých“ osvědčených vzorů.

Návod jak postupovat nám dávají ověřené postupy – vypracované metodiky.

Metodiky odrážejí určité **náhledy** na „realitu“, říkají „jaké“ kroky učinit v jakém pořadí a „jak“ je provádět. Dobré metodiky nám říkají i „proč“ to tak má být.

Metodiky jsou konservovanou zkušeností několika generací programátorů a projektantů. Zobecnění principů, zásad, které se osvědčily, viz historie UML.

Místo abychom se snažili popsat systém jako celek, vytváříme na něj jednotlivé pohledy – jeho jednotlivé, dílčí modely. Díváme se na systém postupně z jednotlivých „míst pozorování“, z jednotlivých perspektiv.

Díváme-li se na systém z jednoho místa, **opomíjíme** vlastnosti z tohoto místa „neviditelné“, nepodstatné a tím si práci zjednodušíme tak, že je mentálně zvládnutelná. Jednotlivé pohledy jsou jednodušší, zvládnutelné.

Opomíjené vlastnosti se neztratí, jsou hlavními vlastnostmi v jiných pohledech - modelech.

Pohledy musíme volit tak, že postupně popíšeme všechny **relevantní vlastnosti systému**. Postupně popíšeme vše, co potřebujeme k dosažení stanoveného cíle.

Z jednotlivých pohledů lze zpětně **zrekonstruovat celý systém** (počítačová tomografie).

Pro tvorbu různých pohledů jsou obvykle využity **diagramy – grafické objekty**, jejichž kombinací lze tyto pohledy vytvářet.

Diagram je graficky znázorněný model. Diagram popisuje jistou část modelu pomocí grafických symbolů.

Tento přístup lze přirovnat k **modelu stavby**, který je tvořen **syntézou dílčích stavebních plánů** odpovídajících specifickým pohledům na stavbu
– plán hrubé stavby, plánu rozvodů elektřiny, plánu rozvodů vody, … V každém z těchto plánů jsou zobrazeny pouze elementy modelu podstatné pro daný pohled, od ostatních elementů modelu je abstrahováno. Pohledy **nejsou nezávislé**, dohromady tvoří **konzistentní pohled na systém**, tedy konzistentní model.

Pro tvorbu modelu systému, respektive pro **tvorbu pohledů na s**ystém, jejichž syntézou bude

model, definuje např. UML **devět typů** diagramů.

Zkušenosti ukazují, že je účelné **tvorbu (návrh a realizaci) IS organizovat do posloupnosti etap**, mluvíme o tzv. **životním cyklu IS**.

Životní cyklus IS není statická sekvence činností. Popisuje dynamiku vývoje, měnící se vnější podmínky (změny legislativy), postup od obecného ke speciálnímu. Životní cyklus IS začíná prvotní představou o systému a končí vyřazením systému z provozu. Samozřejmě není znám přesný a úplný (matematický) model životního cyklu. Nepřesné modely ŽC jsou však nepostradatelné pro řízení projektů.

Základní fáze životního cyklu IS:

- Stanovení globálních cílů, specifikace požadavků, specifikace vlastností, které by měl budoucí systém realizovat (implementovat).
- Analýza systému, tvorba analytického, logického modelu systému, model požadovaných vlastností systému.
- Návrh (design), specifikace způsobu, jak požadované vlastnosti implementovat
- Implementace systému, převedení navrženého systému do spustitelného kódu
- Testování vytvořeného kódu
- Nasazení systému, provoz, údržba

V rámci životního cyklu IS řešíme i řadu organizačních a ekonomických otázek:
proč, pro koho, termíny, cena, pracovní tým, situace na trhu, návratnost investice, řízení pracovního týmu, hardwarové prostředky, dokumentace, reakce na změny.

Model vodopád
Pro řízení a vývoj IS by bylo ideální, kdyby po úplném ukončení jedné fáze, etapy ŽC následovala další a k předchozí etapě by nebylo nutné se již vracet.

Model vodopád je složen z posloupnosti vymezených činností.

Realita je však díky své složitosti jiná. Jednotlivé fáze, etapy ŽC se překrývají. Tak jak postupujeme v ŽC, postupně upřesňujeme výchozí poznatky a musíme se vracet k předchozím etapám. V průběhu práce se také mění výchozí požadavky uživatelů a vnější (legislativní, technologické prostředí).

Prototypový model
Tento model se začal prosazovat v 80.letech. Jeho hlavním cílem je urychlení vývoje IS využitím prototypů.

Prototyp můžeme chápat jako zjednodušenou implementaci celého systému nebo jako plnou implementaci části systému.

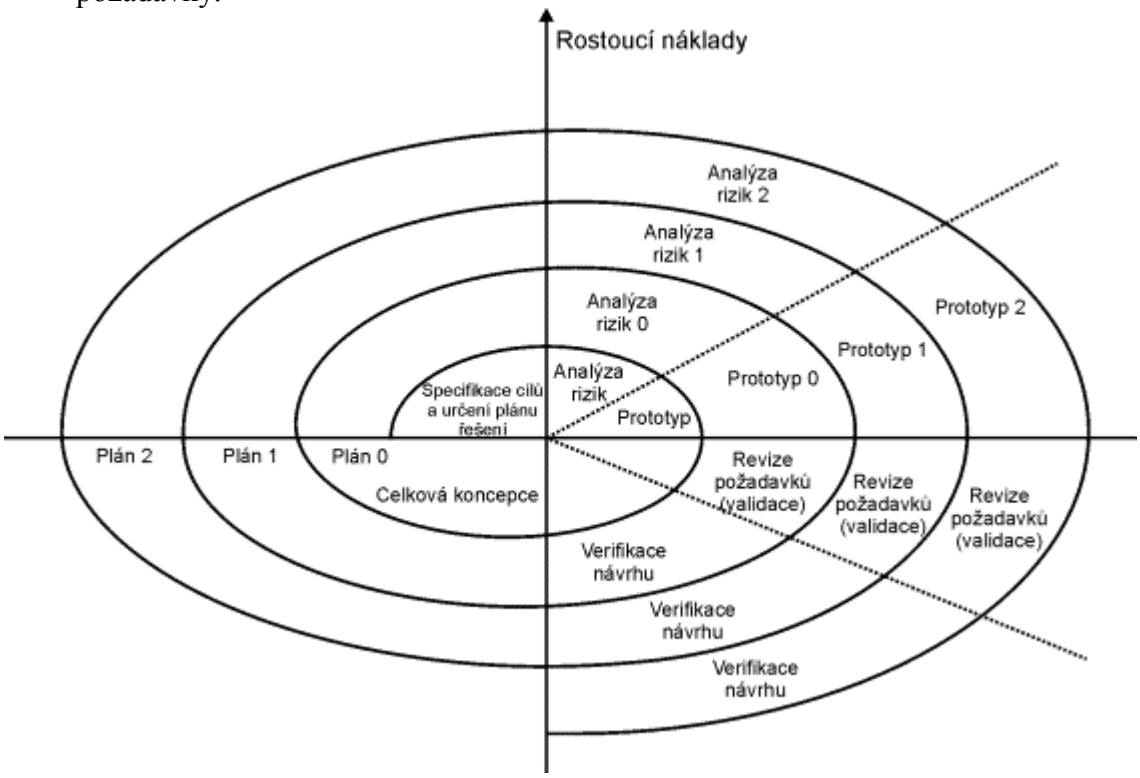
Prototyp je provedena v co nejkratším čase a v takové funkčnosti, která umožní ověřit, otestovat požadované vlastnosti (např. umožňuje zákazníkovi reagovat na výsledky). Na základě vyhodnocení vlastností prototypu jsou upřesňovány požadavky a modifikován další vývoj.

Spirálový model
Tento model vytvořil B.W.Boehm v roce 1988 a je kombinací prototypového přístupu a analýzy rizik.

Základem celého modelu je neustálé **opakování vývojových kroků** tak, že v každém dalším kroku se na již ověřenou část systému přibalují části na vyšší úrovni.

Postup vývoje v jednotlivých krocích se skládá z následujících částí.

- Specifikace cílů a určení plánu řešení.
- Vyhodnocení alternativ řešení a **analýza rizik s daným řešením souvisejících** (je daná alternativa vyhovující, únosná).
- Vývoj prototypu dané úrovně a jeho předvedení a vyhodnocení.
- Revize požadavků neboli validace (testování zda prototyp pracuje tak jak má).
- Verifikace, neboli ověření zda celkový výstup daného kroku je v souladu se zjištěnými požadavky.



Úhlová dimenze modelu reprezentuje postup prací v čase, radiální dimenze pak rostoucí náklady. Nevhodné prototypy jsou opuštěny. Každý nový průchod cyklem rozvíjí nejlepší prototyp.

Nevýhodou modelu je špatný odhad termínu dokončení projektu a jeho celková cena. Proces vývoje je jakoby otevřen.

Strukturální a objektová analýza

Objektový model – diagram tříd

Původní strukturální přístup k analýze IS spočíval v **rozdělení** systému na funkční a datovou část (např. DFD diagramy a ER model).
Přínosem byla funkční hierarchická dekompozice – psaní programu shora dolů a datové konceptuální modelování.

Rostoucí složitost systémů (magická hranice 1000 entit a 10000 funkcí) znemožňuje soudržnost datové a funkční vrstvy. Konstruovali se matice, kde řádky jsou datové entity a sloupce funkce systému. Koexistence se označovala křížkem – možnost dekompozice systém – diagonální matice.

Objektový přístup čelí kromě jiného složitosti systému tím, že třída - objekt jako nositel (funkční) odpovědnosti – dovednosti, plně **odpovídá** za svá data.

Objekt má svou identitu, vlastnosti, chování a **odpovědnost**. Síla odpovědnosti spočívá v tom, že je **nedělitelná** – žádný jiný objekt nemůže odpovědnost sdílet – dělit se o ni, plést se do ní.

Modelování tříd a objektů je **klíčová aktivita** objektově orientovaného vývoje.

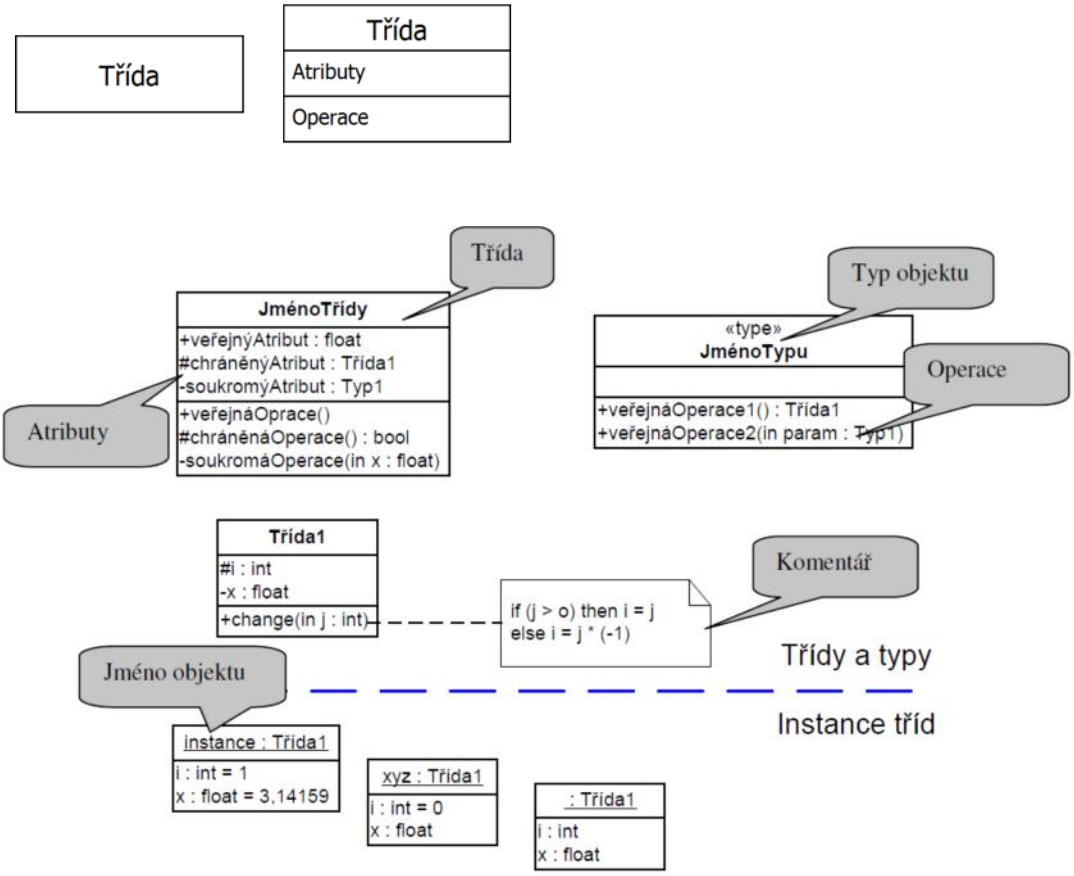
Třída je popisem **množiny objektů** sdílejících stejné vlastnosti - atributy, chování – operace (metody) a vztahy.
Objekt je instancí třídy (chybně se pojem třída a objekt volně zaměňují).

Definice – J. Rumbauh: objekt je diskrétní entita s jasně definovaným rozhraním, které zapouzdřuje stav a chování.

Třídu si můžeme představit jako razítko, objekty jsou pak otisky tohoto razítka, které vidíme na papíře.

Při návrhu třídy neuvažujeme o konkrétním naplnění atributů, pouze určíme jejich název a typ. Teprve při vzniku instance objektu se atributům přiřadí skutečné hodnoty.

Třída je jednoznačně určena svým názvem (v příslušném názvovém prostoru – balíčku). Pro třídu je možno definovat vlastnosti - atributy (Attribute) a chování - operace (Operation).
Vizuálním elementem:



Hledání tříd, jejich atributů a kompetencí - vyberme z reality objekty, kandidáty pro zobecnění na třídy a prověřme jejich vhodnosti:

- Potenciální třída je smysluplná, pokud je nezbytná pro funkci systému.
- Potenciální třída je dostatečně stabilní a invariantní vůči vnějším změnám např. technologie, legislativy apod.

Hledání tříd na základě analýzy podstatných jmen a sloves.

Analyzujeme jazyk problémové domény, např. text sebraných požadavků. Podstatná jména a jejich spojení mohou označovat třídy nebo atributy.

Slovesa mohou označovat odpovědnosti, chování tříd.

Pozor na skryté, utajené třídy, které nejsou v textu uvedeny.
From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Datové modelování, perzistence objektů, konceptuální a fyzický datový model.

Thursday, May 30, 2013 8:24 AM

Datové modelování

- jedna ze základních funkcí IS je ukládání a následné zpracování informací ve formě dat, proto se provádí při návrhu IS také datové modelování
- jeho úkolem je zvolit jaké objekty, jako nositele informace, potřebujeme ukládat do trvalé paměti a jaké jejich vlastnosti a vztahy mezi nimi chceme uchovávat
- při datovém modelování obvykle vytváříme konceptuální a fyzický datový model a také určujeme způsob perzistence objektů

- Cílem datového modelování je navrhnout „kvalitní“ datovou strukturu a databázový systém pro konkrétní IS.

Perzistence objektů

- zabývá se kam a jakým způsobem budou objekty trvale uloženy
- objekty se obvykle ukládají do relační databáze ve formě datových záznamů v tabulkách
- pro programový přístup do databáze se často používá standardizované softwarové API pro databáze jako ODBC nebo JDBC
- pro usnadnění programátorské práce při vytváření perzistentní vrstvy můžeme využít objektově-relační mapování, které nám zajistí automatickou transformaci ukládaných objektů do záznamů relační databáze - např. framework Hibernate pro Java aplikace

Konceptuální datový model

"Konceptuální modelování má své kořeny v počátcích datového modelování a úzce souvisí i například s teorií relačních dat. Jedná se o modelování reality prostřednictvím základních pojmů (konceptů) a jejich vzájemných souvislostí. Konceptuální modelování je dnes široce rozvinutý samostatný obor s propracovanými nástroji a metodami, jejichž principy se odráží i v pravidlech modelování tříd objektů pomocí UML."

- vyjadřuje jaké objekty a jejich atributy budeme ukládat a vztahy mezi nimi
- pro grafické vyjádření se používají ERA modely (diagramy)

Při datovém modelování vytváříme nejprve logický - konceptuální datový model. Konceptuální datový model představuje určité zobecnění oproti implementaci datové struktury v konkrétní relační databázi.

Konceptuální model – fáze návrhu, volby technologie, nezávislý na konkrétní databázi

Fyzický datový model

- odvozuje se z konceptuálního modelu a vyjadřuje navíc jak přesně budou data v konkrétní databázi uložena
- také se popisuje ERA modely, které obsahují i přesné databázové typy atributů tabulek

Zvolíme-li konkrétní databázi (např. Oracle) mluvíme o fyzickém datovém modelu, na který je konceptuální model převeden.

Fyzický model – fáze implementace pro konkrétní databázi, konkrétní implementace. Z fyzického modelu můžeme generovat SQL skripty, případně se napojit přímo na databázi. Na fyzické úrovni můžeme psát uložené procedury a triggery.
Dva přístupy:

- **Při tvorbě konceptuálního datového modelu vycházíme z diagramu analytických, resp. návrhových tříd s jasnou představou o požadavcích na perzistenci.Postup od objektové analýzy, přes návrhové třídy k implementaci.**

Primárně pracujeme s konceptuálním modelem, objektová nástavba je sekundární.

ERA modely

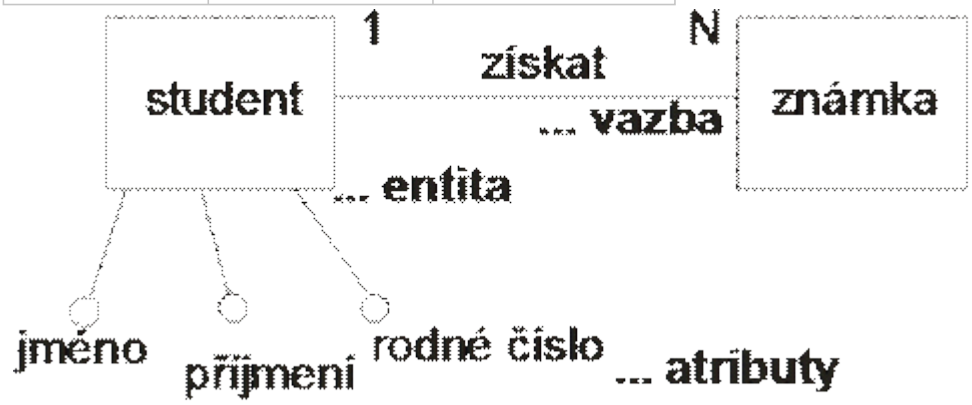
viz DB1 - vypsáno z DB1

Entita – model z reálného světa

Relation – podchycuje vztahy mezi entitami

ERA modely = modelová analýza

Datový model	Datová struktura	Souborový přístup
Entitní množina	Tabulka	Soubor
Entita	Řádka	Záznam
Atribut	Název sloupce	Položka



Vazby

1:N - vyjadřuje, že jedné entitě E1 může příslušet více entit z entitní množiny E2

jedné entitě z E2 přísluší jen jedna entita z E1

nejlépe 1:N(0)

př.: student – známka

1:1 - na vazbě se podílí jej jedna entita z E1 a jedna z E2

vždy se ptát, proč je to rozdělené, proč to není jedna entita

vazba je zajímavá, pokud alespoň jeden konec volný (nepovinný)

př.: student – známka (student nemusí mít známku)

N:N - jedné entitě z E1 přísluší více entit z E2

jedné entitě z E2 přísluší více entit z E1

př.: student – rozvrhová akce

ER modely

- primární – minimální množina atributů, která jednoznačně určuje entitu
- na vazbu se díváme jako na entitu – lze k ní přidat atributy (čtenář – výpůjčka – exemplář)
- vazba 1:N se při realizaci vytvoří tak, že do "podřízené" tabulky přenesu klíč (cizí klíč) z "nadřazené" tabulky
- vazba M:N nelze realizovat, nelze vyřešit pomocí cizích klíčů = musí se provést rozklad vazby
- mezi dvěma entitními množinami může existovat více vazeb
- vazba nemusí být binární, ale může být n-ární
- vazba může být i unární (sama na sebe)

Některé datové modely rozlišují entitní množiny regulární a slabé

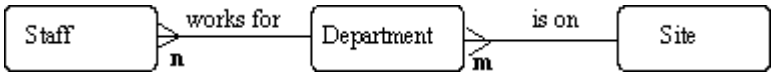
Slabá entitní množina je taková, u níž nelze určit, či nemůžeme zjistit nadřazenou množinu

FAN

Problém: 2 vazby 1:N se větví z jedné entity

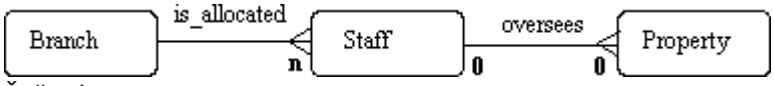
datové položky ve dvou složkách nejsou v přímém vztahu, ale mají vazbu založenou na datových položkách ve třetí složce

Řešení:

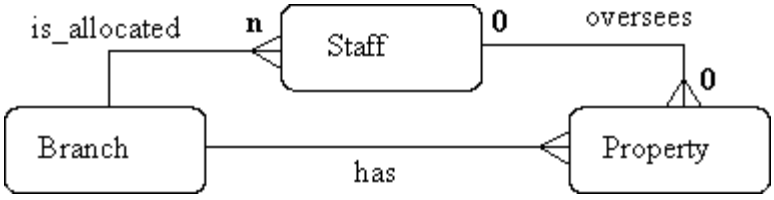


CHASM trap

Problém: Existuje vztah mezi entitami, ale chybí vazba.



Řešení:



Objektově relační mapování - Perzistence

Objektově relační mapování – O/R slouží k tomu, aby bylo možné snadno používat relační databáze v prostředí objektově orientovaných programovacích jazyků.
Vzhledem k tomu, že objektově orientovaný návrh dat není jednoznačně převoditelný na relační databáze a opačně, používají se různé formy mapování.
Mapování má za účel načítat data z relační databáze a naplnit jimi příslušné datové položky objektů včetně vazeb mezi objekty, případně naopak datové položky objektů ukládat do databáze.
Snahou ORM je co nejlepší využití obou zmíněných technologií

- objekty by měly reprezentovat objekty reálného světa, jak to požadují principy OOP
- na straně databáze bychom zase měli využít všech možností relačních databází – indexy, pohledy, primární klíče, triggery a uložené procedury.

Alternativou k O/R mapování je použití objektové databáze, která je navržená přímo pro ukládání objektů. Použití takové databáze eliminuje potřebu převádět data z objektové podoby do relační. Data jsou uložena přímo ve své objektové reprezentaci.
Objektové databáze zatím nejsou příliš rozšířené.
V současnosti je nejpoužívanějším nástrojem pro ORM produkt od firmy JBOSS Hibernate.
Hibernate je O/R mapovací nástroj pro jazyk Java. Jde o volně šiřitelný open source software.
Nabízí prostředí pro mapování objektového modelu na tradiční relační schéma.

Perzistentní třídy musí splňovat jisté vlastnosti, obsahovat

- konstruktor bez parametrů
- getter a setter metody pro perzistentní položky

Konstruktor bez parametrů Hibernate používá při načítání objektu z databáze. Jeho zavoláním v paměti vytvoří prázdný objekt, jehož položky následně nastaví podle hodnot uložených v databázi pomocí setter metod. Getter metody Hibernate používá při čtení položek při ukládání objektu do databáze.

Základní mapování - mapování tříd na tabulky
Perzistentní (entitní, bussines) třídy, resp. jejich instance - objekty odpovídají entitám konceptuálního datového modelu, resp. řádkům tabulek fyzického modelu. Atributy třídy se stanou sloupci tabulek.

Mapovací soubory pro Hibernate se píší v jazyce XML nebo se využívá tzv. anotací Javy (zápis metadat přímo do kódu). Každá třída se mapuje na jednu tabulku. Každá tabulka musí obsahovat primární klíč.
Mapovací soubor obsahuje výčtem elementů <property>, které reprezentují položky, které mají být ukládány a jak mají být ukládány.

Způsob uložení položek lze ovlivnit velkým množstvím atributů. Několik nejpoužívanějších popisuje následující výčet.

- column="column_name" - určuje název sloupce. Implicitně se používá název proměnné.
- type="typename" - určuje typ konverze mezi Java typem a SQL typem.
- not-null="true|false" určuje zda je možné do daného sloupce uložit NULL hodnotu.

Mapování atributů musí odpovídat konverzi datových typů, norma SQL – 92 definuje standardy datových typů (opakem jsou transientní třídy).

Mapování vztahu
Mapováním vztahů zajišťujeme, že se může mezi objektovým modelem a databází současně „přenášet“ síť vzájemně provázaných – asociovaných objektů. Základní výhoda ORM
Při použití Hibernatu jako ORM vrstvy se vazby dělí ještě na jednosměrné a obousměrné. Rozdíl je v tom, že u jednosměrné vazby má referenci jen jedna entita. Druhá tedy žádnou referenci nemá, kdežto u obousměrné vazby mají reference obě entity.
Vztah se v mapovacím souboru mapuje pomocí jednoho z elementů

- <one-to-one>
- <one-to-many>
- <many-to-one>
- <many-to-many>

podle kardinality daného vztahu. Jediným povinným atributem je name, ostatní atributy jsou nepovinné. Nepovinné atributy jsou podobné jako u elementu <property>.

Mapování dědičnosti
Protože cílový fyzický datový model nepřipouští dědičnost tabulek, viz norma SQL 92, existují tři možnosti:

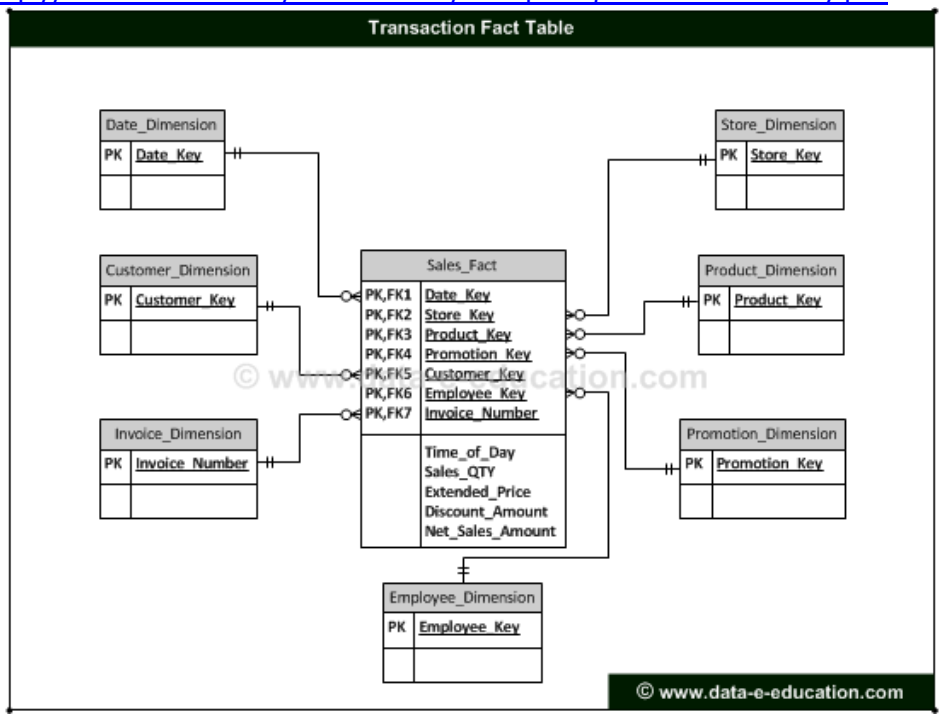
- mapování 1:1 – každá třída se mapuje do samostatné tabulky, jedna instance objektu je rozložena po více tabulkách, řada nevýhod.
- zahrnutí do nadtřídy – atributy podtříd jsou zahrnuty do nadtřídy, z třídy a jejich podtříd vnikne jedna tabulka. Vhodné v případě malého počtu podtříd.
- rozpuštění do podtříd – všechny atributy nadtřídy jsou přeneseny do tabulek pro všechny podtřídy. Počet tabulek odpovídá počtu podtříd. Vhodné pro velký počet podtříd.
  Viz CASE

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

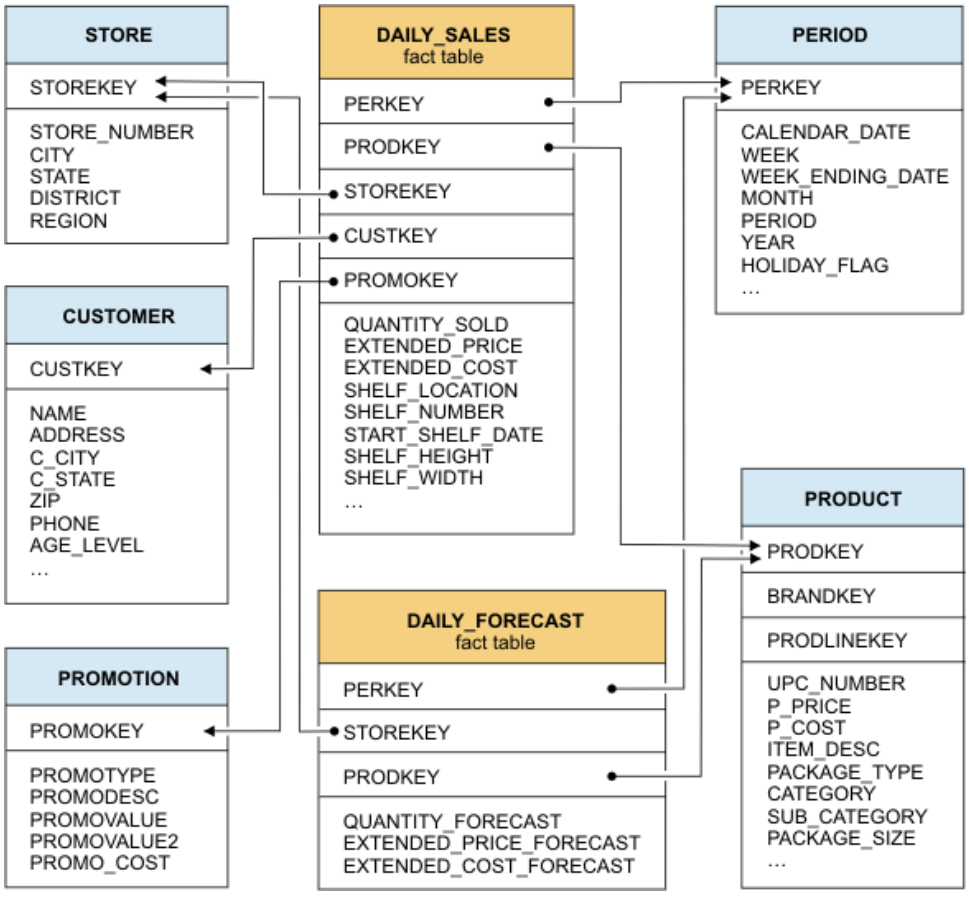
OLAP systémy, jejich význam a oblasti využití, základními principy, dimenze, agregace, extrakce a transformace dat, srovnání transakčních a analytických systémů (OLAP a OLTP technologií).

Thursday, May 30, 2013 8:24 AM

http://www.common.cz/attachments/118_petr_jasa_datove_sklady.pdf



From <http://www.bing.com/images/search?q=table+of+facts+table+of+dimensions&view=detail&id=CDCE4349212949F443E8EC07EA739665F19DCA98&FORM=IDFRIR>



From <http://www.bing.com/images/search?q=table+of+facts+table+of+dimensions&view=detail&id=9E9783F312C8C26698296EA1AFD9230AB0B4D1DB&FORM=IDFRIR>

Dimenze

A Dimension is a structural attribute of a cube that is a list of related names-known as Members-all of which belong to a similar category in the user's perception of a data. For example, all months, quarters and years make up a Time dimension; likewise all cities, regions and countries make up a Geography dimension. A Dimension acts an index for identifying values within a multi-dimensional array and offers a very concise, intuitive way of organizing and selecting data for retrieval, exploration and analysis. Dimensions are the business parameters normally seen in the rows and columns of a report.
Dimensions are lists of related terms used to organize your data. Thus, a natural Dimension name for the Members January, February and March might be Months. Dimensions, in turn, are used to construct Cubes, the multidimensional structures in which you store and model data.

From <http://olap.com/w/index.php/Dimension>

Význam a oblasti využití

BI - Business intelligence - pro podporu rozhodování, analýzy historických dat, procesní ukazatele

Extrakce, Agregace = asi ETL

- Extract Transform Load, způsob plnění datového tržiště daty. Data jsou vybrána z relačních databází kde se nachází (Extract), vyčištěna, upravena (Transform) a pak je s nimi naplněno datové tržiště (Load).

Základní problémy u běžných transakčních databázových systémů:

- nedosažitelnost dat skrytých v transakčních systémech
- dlouhá odezva při plnění komplikovaných dotazů
- složitá, uživatelsky nepříjemná rozhraní k databázovému softwaru
- cena v administrativě a složitost v podpoře vzdálených uživatelů
- soutěžení o počítačové zdroje mezi transakčními systémy a systémy podporujícími rozhodování

Cesta k řešení těchto problémů = datové sklady, tzv. Data Warehouse – DW

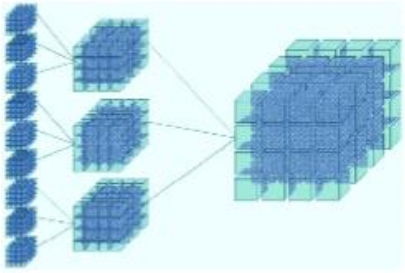
Datawarehouse

- samostatný informační systém postaven na již pořízených datech, určen především k jejich analýze
- architektura založená na relačním SŘBD, která se používá pro údržbu historických dat získaných z databází operativních dat, jenž byla sjednocena a zkontrolována před jejich použitím v databázi DW
- data z DW jsou aktualizována v delších časových intervalech, jsou vyjádřena v jednoduchých uživatelských pojmech a jsou sumarizována pro rychlou analýzu
- DW je obrovská databáze obsahující data za dlouhé časové období
- často slučuje data z více rozdílných zdrojů, které mohou obsahovat data různé kvality nebo používat nejednotné formáty a reprezentace
- objemově zabírá stovky GB až několik TB
- nemusí být databází v běžném smyslu, tj. pro přesné provádění transakcí
- je určen pro rychlé vyhledávání
- nejsou kladeny nijak důrazné požadavky na správnost a úplnost dat

Charakteristika

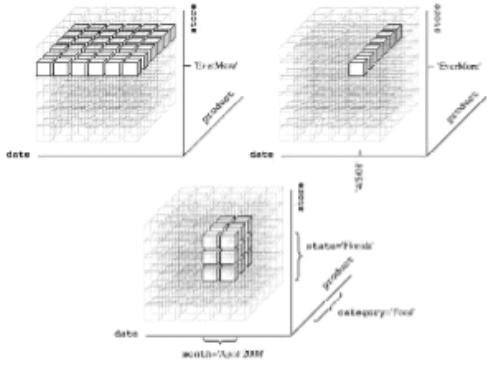
- data jsou uložena na různých místech ve formě relačních tabulek
  - uživatelé mohou tabulky jen číst
  - zapisovat může aktualizační program pravidelně udržující tabulky
- dotazy jsou většinou komplexní
  - podporují tzv. on-line analytické zpracování (OLAP)
    - výrazně se liší od on-line transakčního zpracování (OLTP)
    - operační databáze je přizpůsobena pro podporu OLTP
      1. složité OLAP dotazy by vyústily do nepřijatelné odezvy
    - typické OLAP operace
    - **roll-up** (sumarizace dat napříč dimenzí)

Roll-up



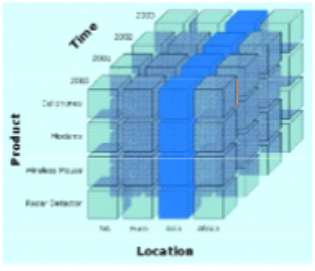
- **drill-down** (zanoření se do nižší úrovně dat pro získání více detailů)
- Drill-up (opak drill down, přechod o level výše pro skrytí detailů a získání lepšího celkového přehledu o datech)

Drill down

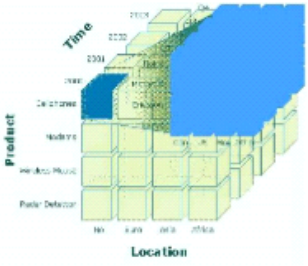


- **slice-and-dice** (selekce a projekce)
- **Slice** - v jednom rozměru krychle zvolíme pouze jednu hodnotu, což vytvoří novou krychli, která je "řezem" té původní
- **Dice** - vybereme konkrétní hodnoty v každé dimenzi krychle, vznikne menší krychle

Slice

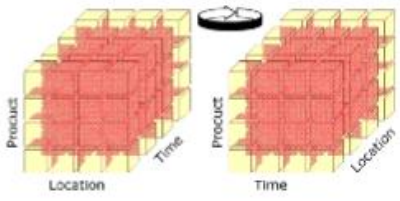


Dice



- **Pivot** (přeorientování vícerozměrného pohledu na data, prostě prohození os)

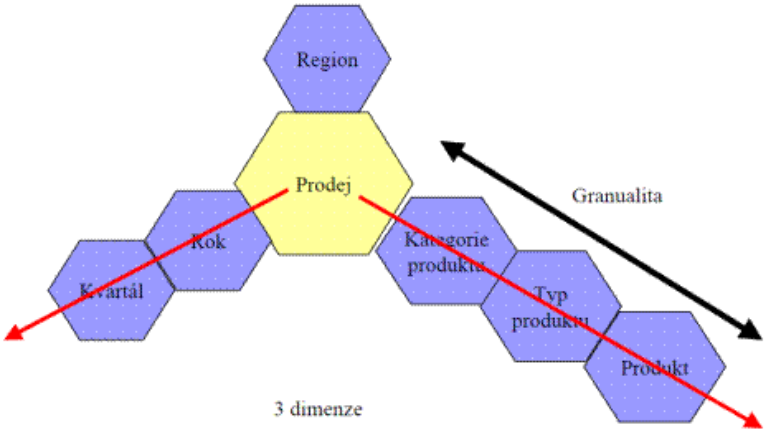
Pivot



- na základě dotazu se pospojují potřebná data do vícerozměrné tabulky (nebo více tabulek), do kterých lze klást SQL dotazy
- pro častější dotazy si uchovávají předem připravené vícerozměrné tabulky
- zátěž je většinou způsobena složitými dotazy, jež přistupují k miliónům záznamů a provádějí

množství operací
- data bývají modelována vícerozměrně
  - v obchodním data warehouse mohou těmito rozměry být např. čas prodeje, místo prodeje, prodavač, výrobek, …
  - rozměry mohou být i hierarchické např. čas prodeje jako den-měsíc-čtvrtletí-rok, zboží jako výrobek-kategorie-průmysl
  - spojení více tabulek pomocí odkazu na řádky jednotlivých tabulek
  - používají speciální organizaci dat, přístupové a implementační metody, jež obecně nejsou v komerčních databázových systémech určených pro OLTP podporovány

Základní myšlenka multidimenzionálního modelování



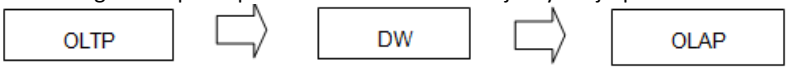
Databázový systém – OLTP (Online Transaction Processing Systems)
- zákaznicky orientovaný
- aktuální data -- lze považovat i za slabinu, při výpadku (chybě), vznikají ztráty pro byznys
- ER schéma
- sofistikované atomické transakce i přes několik systémů(bank, po síti,...)
- velikost DB až několik GB
- jednoduché a efektivní
- příkladem je bankomat

DataWarehouse – OLAP (Online analytical Processing)
- orientovaný na trh, rychlé (oproti OLTP) získání výsledků na analytické dotazy
- historická data, multidimenzionální datový model
- agregovaná data (nenormalizovaná=redundantní)
- schéma hvězdy či vločky
- převážně pouze čtení
- velikost až TB
- použití: byznys reporty o prodeji, marketing, management reporty, rozpočty, finanční předpovědi a reporty

Použití DW
- prezentace dat
- testování hypotéz
- objevování nových informací

Architektura DataWarehouse
- tři úrovně:
  - klient
  - OLAP server (MOLAP/ROLAP server)
  - databázový server DW
- data lze organizovat v tzv. multidimenzionálním datovém modelu
  - odlišný od modelu relačního
  - odpovídá mu specializovaný software, multidimenzionální SŘBD (MDD)
  - model připomíná techniku spreadsheet ve více než dvou rozměrech
  - data jsou implementována pomocí vícerozměrných polí, jejichž dimenze odpovídají dimenzím podnikání organizace
- navržení a vytvoření DW je proces skládající se z následujících bodů:
  - definovat architekturu, umístění a rozčlenění dat a fyzickou organizaci
  - naplánovat kapacitu, vybrat OLAP servery a nástroje
  - spojit servery, klientské nástroje, zdroje přes gatewaye, drivery ODBC, …

- navrhnout schéma a pohledy, přístupové metody, některé složité dotazy
- mít skripty pro získávání, čištění, transformaci, ukládání a aktualizaci dat
- vytvořit koncové uživatelské aplikace
- spustit data warehouse i aplikace

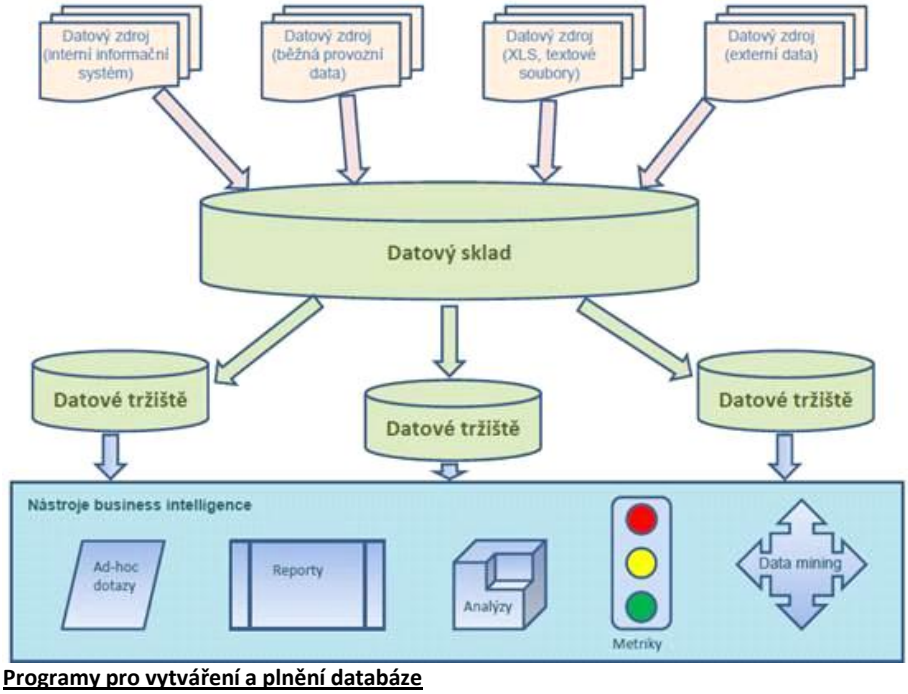
- vytvoření je složitý proces trvající mnohdy i několik let
- mnoho organizací proto používá Data Mart umožňující rychlejší práci
- 

Datová tržiště (Data Mart)

- DW slouží jako základna pro extrakci množin dat, resp. jejich agregaci do dílčích (replikovaných) MDD (Multidimenzionální DB)
  - MDD může pro DW sloužit ve dvou rolích
  - ”front-end” pro DW a poskytovat uživateli služby pro realizaci analytického zpracování (DW/OLAP)
  - “front-end” jednomu (několika) systémům OLTP - alternativa za DW, tj. poskytnout uživateli s OLTP data analytickým způsobem (OLTP/OLAP) – jde vlastně o datové tržiště

Systém OLAP (OnLine Analytical Processing)

- na databázové stroje jsou kladeny specifické požadavky
  - objem zpracovávaných dat
    - transakční systém o velikosti gigabajtů dosáhne použitím jen jedné dimense velikosti desítek či stovek gigabajtů
  - rychlost odezvy analytického systému je důležitá
  - počet uživatelů současně pracujících s databází není zajímavý
    - počet pracovníků vyššího managementu je omezen
    - pro pracovníky nižších stupňů bývají údaje z datových skladů převedeny do menších specializovaných databází – datových tržišť
- s těmito omezeními se vyrovnává dvojím způsobem
  - uzpůsobení stávajících systémů pro práci s vícerozměrovými daty
    - přidáním modulu, který to zajišťuje a prostředků pro jeho ovládání
    - v lepším případě mění způsob uložení dat, v horším “překládá” operace s vícedimenzionálními daty na operace s daty relačními
  - vytvoření speciálního systému správy dat, určeného pouze pro OLAP
    - umožňuje provést maximum optimalizací vzhledem k nárokům kladeným analytickým způsobem práce - převažující způsob



Programy pro vytváření a plnění databáze

- převodní programy
  - načtení data z několika databází, či souborů a udělat z nich novou databázi, agregace se musí naprogramovat
- systémy znázorňující převodu dat graficky a administrátor dat namapuje zdrojová data do struktur vytvářeného datového skladu
  - výsledkem jsou buď programy (scripty) nebo přímo vykonání funkce
- moduly pro plánování jednotlivých akcí

Nástroje pro práci s daty - poslední trendy v architektuře klient/server

- nabízejí variantu tenkého klienta v podobě HTML prohlížeče

Reporting, monitorování, ad-hoc dotazy

- programy umožňující kladení dotazů a formátování odpovědí
  - nejčastěji jde o vizuální dotazovací nástroje
  - makra v tabulkovém procesoru
  - uživatelské rozhraní různě propracované:
    - zadání seskupení výsledku podle různých kriterií
    - formální kontrola dotazů
    - vytváření slovníků a metadat

MOLAP - Multidimenzionální OLAP

- datová krychle (obsahuje fakta)
- hierarchické dimenze (částečné či totální uspořádání)
  - vločkové schéma -- hlavní tabulka faktů je v relaci s dimezionálními tabulkami, přes cizí klíče, dimenzionální tabulky mohou být také v relaci s dalšími subdimenzionálními tabulkami podobně jako hlavní tabulka faktů; vytváří hierarchie dimenzí
  - hvězdové schéma -- je speciální případ vločkového, dimenzionální tabulky již nejsou v relaci s dalšími subdimenzionálními tabulkami; žádné hierarchie, jednodušší

Pozn: dle mého názoru do MOLAP patří jen multidimenzionální krychle (proto MOLAP). Vločka a hvězda jsou ROLAP.

ROLAP – Relační OLAP

- na relační architektuře založený model DW strukturou propojených DB tabulek - Relační OLAP (ROLAP) – pomalejší zpracování než MOLAP
- užívá relační nebo rozšířený relační DBMS, např server METACUBE Informix, pracuje s relačními tabulkami uspořádanými do hvězdy/vločky, adresuje pomocí klíče, data jsou neagregovaná)

Srovnání OLAP a OLTP

Znak	OLTP	OLAP
Charakteristika	Provozní zpracování	Informační zpracování
Orientace	Transakční	Analytická
Uživatel	Běžný uživatel, databázový administrátor	Znalostní pracovník (manažer, analytik)
Funkce	Každodenní operace	Dlouhodobé informační požadavky, podpora rozhodování
Návrh databáze	Entitně-relační základ, aplikačně orientovaný	Hvězda/sněžná vločka, věcná orientace
Data	Současná, zaručeně aktuální	Historická
Sumarizace dat	Základní, vysoká podrobnost dat	Shrnutá, kompaktní
Náhled	Detailní	Shrnutý, multidimenzionální
Jednotky práce	Krátké, jednoduché transakce	Komplexní dotazy
Přístup	Číst, pořizovat a aktualizovat	Pouze číst
Zaměření	Vkládání dat	Získávání informací
Počet dostupných záznamů	Desítky	Miliony
Počet uživatelů	Stovky – tisíce.	Desítky – stovky.
Velikost databáze	100 MB až GB	100 GB až TB
Přednosti	Vysoký výkon, vysoká přístupnost	Vysoká flexibilita, nezávislost koncového uživatele
Míry hodnocení	Propustnost transakcí	Propustnost dotazů a doba odezvy

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Vlastnosti a typy CASE nástrojů a jejich význam v analýze a návrhu informačních systémů.

Thursday, May 30, 2013 8:24 AM

CASE - obecně
CASE – Computer Aided Software Engineering – jsou programy určené k tomu, aby **podporovaly vývoj informačních systémů**.

Společný pohled - používání CASE nástrojů umožňuje designerům, programátorům, testerům a manažerům (tedy všem, kteří se na vývoji systému podílejí) mít **společný náhled** na to, jak projekt vypadá jako celek, jak jeho jednotlivé části v detailu a jaký je jeho stav v jednotlivých fázích vývoje.

CASE:
- **pomáhá** zajistit to, že proces vývoje projektu je řízený, řiditelný a kontrolovatelný.
- **čelí složitosti** systému, která by bez jejich pomoci byla těžko zvládnutelná.
- **zajišťuje kvalitu procesů** vývoje softwaru (díky použitým metodikám a odhalování chyb při jejich použití)**.**
- **zajišťuje** značnou **úsporu času** (a tedy nákladů) potřebného k vývoji systému.
- **slouží jako úložiště** projektové dokumentace.

Některé CASE nástroje jsou přímo integrovány do vývojových prostředí.

Odhady hovoří o tom, že použití CASE (přes počáteční zpomalení) nástrojů představuje úspory okolo **50 až 70 procent** v dalších etapách životního cyklu softwaru.

CASE nástroje jsou založeny na dvouvrstvé architektuře.
Základ každého z nich tvoří tzv. „**repository“**, kam se ukládají veškeré informace o navrhovaném systému (jedná se o databázi, která automaticky udržuje data v konzistentním stavu).
Nad společným repository pracuje druhá modelová vrstva, která zpřístupňuje informace uložené v repository.

Každá z modelových vrstev se opírá o jistou metodiku a reprezentuje jistý pohled na informace uložené ve společném repozitury. Jednotlivé modely jsou díky společnému repozitury na sebe vzájemně převoditelné (např. z diagramu tříd můžeme vygenerovat fyzický datový model).

Vývoj a typy CASE nástrojů

- CASE systémy vznikly v **sedmdesátých letech dvacátého století**, v situaci, kdy začala prudce narůstat složitost IS.

- CASE se na trhu začaly výrazně prosazovat zhruba *v* **polovině osmdesátých let**.
- Tyto systémy vznikly v okamžiku dosažení kritické kvality v metodách, organizaci práce a technologiích, potřebných při vývoji informačních systémů. Od podpory čistě vývojových fází životního cyklu IS (analýzy a konstrukce systému) k podpoře strategických rozhodování na počátku projektu a operativních činností souvisejících s provozem systému a řízením jeho změn a rozvoje.

- To, co dalo podnět ke vzniku CASE nástrojů a určuje také směry dalšího vývoje, je **metodikou tvorby informačních systémů – strukturální a objektové metodiky.**

- Postupem doby a s měnícími se požadavky dnes existuje celá řada CASE nástrojů. Je to díky podporovaným metodikám a samozřejmě také tím, v **jaké fázi vývoje je nástroj používán.**

- **Cílem je využít CASE ve všech fázích životního cyklu IS od s**pecifikace požadavků, analýzu, návrh a kódování po údržbu IS.

Kategorie CASE nástrojů

Podle podporovaných fází životního cyklu systému lze CASE nástroje rozdělit do dvou základních kategorií:

- **Integrované CASE**. Zaměřují se na podporu celého životního cyklu vývoje IS.

- **Specializované CASE**. Tyto nástroje jsou orientované na určité specifické etapy.

Specializované CASE nástroje

Nástroje používané v různých etapách se liší. Většinou pokrývají jen určité činnosti. Podle životního cyklu vývoje lze CASE nástroje rozdělit dle na:

- ***Pre CASE*** –
  Tyto nástroje jsou určeny pro tvorbu celkové strategie IS.

- ***Upper CASE*** –
  Nástroje této kategorie podporují plánování, specifikaci požadavků, modelování organizace podniku a celkovou analýzu IS**.**

  Hlavním úkolem je analýza organizace, zachycení všech procesů, definice klíčových toků a dokumentace zjištěných požadavků. Použití je pro specifikaci cílů a počátečních požadavků a řízení projektu.

- ***Middle CASE*** –
  Tento druh CASE nástrojů je základem všech komerčně dodávaných nástrojů.

  Prostředky této kategorie slouží pro podrobnou specifikaci požadavků analýzu a návrh systému. Používají se také pro dokumentaci a vizualizaci systému.

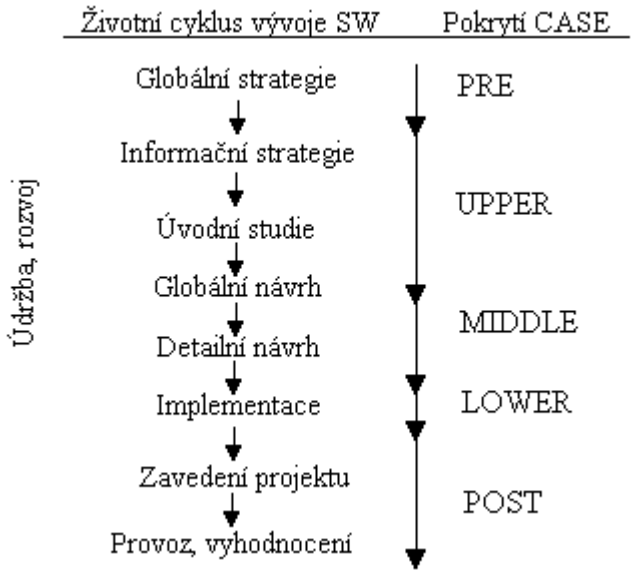
- ***Lower CASE*** –
  Tyto nástroje slouží především jako podpora kódování, testování a údržby. Jejich součástí jsou i **nástroje forward engineeringu**[1], tedy generátory kódu z modelu (ty mohou generovat kostru nebo podstatnou část výsledného kódu, programátor poté doplňuje jen nutné detaily a algoritmy). Dále pak jde o prostředky pro **reverse engineering**[2], které umožňují získat model z již existující aplikace, prostředky pro plánování a zjišťování kvality SW (sběr informací o testování, vyhodnocení testů, řízení testování), pro správu konfigurace, prostředky pro sledování a vyhodnocování práce systému. Funkce CASE nástrojů této kategorie se často překrývají s funkcemi obecných vývojových prostředí.

- ***Post CASE –***
  Tento druh CASE nástrojů podporuje organizační činnosti jako zavedení, údržbu a rozvoj IS.

Působnost takto rozdělených CASE nástrojů se překrývá, protože jimi podporované činnosti se mohou vyskytovat v různých fázích životního cyklu vývoje IS.

Fáze životního cyklu IS a CASE nástroje

Vztah CASE nástrojů a fází životního cyklu IS je zachycen na následujícím obrázku:



Pravdivé a mylné představy o CASE nástrojích

Pravdivé představy:

- Hlavním přínosem těchto nástrojů je vytváření úplných podkladů pro programování aplikací.
- CASE jsou nástroje, které mohou zlepšit produktivitu práce, efektivita práce vždy závisí na osobních kvalitách jednotlivých pracovníků.
  - Mohou generovat části kódu, ale nenahrazují programovací jazyky.
  - Praxe ukázala, že CASE nástroje často selhávají právě díky nedisciplinovanosti uživatelů.
- Automatizací „chaosu“ vznikne automatizovaný „chaos“.
  - Na počátku práce je nutné vykonat velmi mnoho činností, jejichž výsledek není dlouho vidět.
  - Dostanou-li stejný CASE dva systémoví analytici, dospějí k dvěma naprosto odlišným řešením.

Mylné představy:

- CASE nástroje slouží jako náhrada programovacích jazyků.
- Všechny CASE nástroje pracují podobně (poskytují stejné výstupy).
- Užívání CASE nástrojů zlepší práci manažerů organizace využívající výsledný produkt.
- CASE odstraňuje potřebu disciplíny a přísného vývoje aplikací IT.
- Od CASE nástrojů se často očekává jako výstup tvorba aplikačního programového vybavení.
- Produktivita dosažená pomocí CASE je okamžitě zřejmá.
- Užívání CASE zaručí konzistenci výstupů.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Zavedení a struktura operačního systému.

Wednesday, May 29, 2013 4:49 PM

Zavedení OS

BIOS

- program přítomný ve vestavěné paměti HW (většinou na základní desce)
- provádí testy a nastavení HW
- vybere zaváděcí jednotku
- načte první sektor (MBR), kde je umístěn program zavaděče a provede skok na adresu jeho programu, čímž mu předá řízení

Zavaděč (bootstrap program)

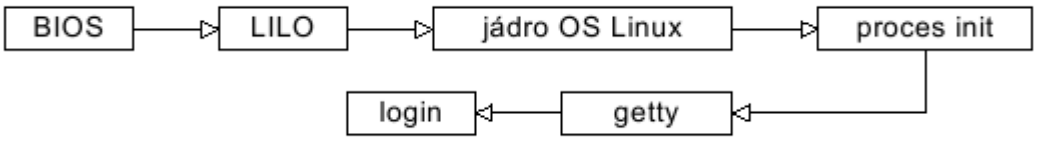
- pro Linux LILO (LInux LOader) je všeobecně použitelný zavaděč (boot loader) pro Linux) nebo GRUB
- dává možnost zvolit startující OS
- načte jádro operačního systému do paměti a spustí ho

Jádro OS

- detekuje hardware a odpovídajícím způsobem nastaví ovladače zařízení
- připojí kořenový svazek pro čtení a provede kontrolu souborového systému
- spustí na pozadí proces init

Proces init

- proces init se konfiguruje pomocí souboru /etc/inittab
- inicializuje operační systém
- spuštěn po celou dobu běhu operačního systému a ošetřuje některé události (úklid v adresáři /tmp)
- spustí služby - Démony
- nakonec spustí program getty pro terminály a virtuální konzoli a v ní program *login*
- nastavením parametru jádra tzn. runlevel lze upravit chování systému



Úrovně běhu systému – runlevel

- runlevel 0 – zastavení systému - halt
- runlevel 1 – jednouživatelský režim Single user mode
- runlevel 2 – víceuživatelský režim bez podpory sítě Multiuser, without NFS
- runlevel 3 – víceuživatelský režim s podporou sítě Full multiuser mode
- runlevel 4 – není použit unused
- runlevel 5 – víceuživatelský režim s podporou sítě a XFree X11
- runlevel 6 – restart systému reboot

Zavádění systému

- nejprve proběhne úspěšný test zavádění systému – POST
  - kontroluje HW v zařízení
  - série testů ke zjištění, zda HW pracuje správně
  - zjištěné chyby jsou uloženy nebo oznámeny – blikáním LED/série pípnutí
  - Po dokončení je řízení předáno bootovací sekvence volající ovládací SW nebo zavaděč OS

- **Bootování**
  - Najde a zavede (vygeneruje se přerušení 19h) se tzv. Bootovací sector (boot sector)
  - **Boot sector -** oblast 512 bajtů na záznamovém médiu, které je jako první nastavené v paměti BIOSu
    - Bootsektor se nachází na prvním sektoru záznamového média (v případě pevných disků je to válec 0 hlava 0 stopa 0 sektor 1)
  - BIOS se snaží najít na tomto sektoru Master Boot Record (MBR) – hlavní spouštěcí záznam.
    - Ten nahraje do paměti na adresu 0000:7C00 a v případě úspěchu mu předá řízení.
    - V případě chybného MBR se bootovací proces přeruší pomocí softwarového přerušení 18h – vygeneruje chybové hlášení (např. NO ROM BASIC – SYSTEM HALTED)
    - Správnost MBR BIOS zjišťuje pomocí kontrolní hodnoty umístěné na posledních dvou bajtech sektoru - **AA55h** (zápis je uložen ve formátu little endian)
    - MBR – ze dvou částí – **partition loader** a **partition tabulky**
    - MBR uchovává záznamy o rozdělení disku (oddílech) a určuje, ze kterého z nich se má bootovat.
    - Pokud MBR OK – řízení se předá partition loaderu
    - **Partition loader** v Partition tabulce vyhledá oddíl, který je označen jako aktivní a přejde na první sektor tohoto oddílu. MBR sám sebe překopíruje na jiné místo v paměti a na své původní místo zkopíruje tento první sektor a předá mu řízení
-

Spuštění počítače

- Po zapnutí PC jsou všechny procesory v reálném režimu
- - Náhodně se vybere jedno jádro -> bootstrap processor (BSP)
- - ostatní CPU jsou nyní application processors AP - pozastavené, dokud je nezapne kernel
- - nyní je BSP v tzv. real mode, vypnuté stránkování (BSP simuluje staré 8086 ze let okolo '78)
- - v tomto stavu je adresováno pouze 1MB paměti bez ochrany (lze v ní spustit cokoliv)
- - u Intel CPU je hack, kdy se nastaví bázová adresa (jako offset) na tzv. reset vector – 0xFFFFFFF0 (konec 4GB paměti - 16B)
- - na adrese reset vectoru je jump na adresu, kde je namapovaný BIOS entry point (zajišťuje základní deska)
- - tento skok vymaže Intelí hack bázovou adresu
- - oblasti v paměti jsou zaplněna správnými daty díky memory map v chipsetu
- - nyní BSP spustí BIOS -> Power-On self test (POST) -> error = pípání PC speakeru nebo zombie PC
- - po POST bootování systému, umístění volitelné (disketa, DVD ROM, HDD, ...)
- - BIOS nyní přečte první sektor umístění (HDD) o velikosti 512B (zero sector) = Master Boot Record (MBR)
- - MBR obsahuje:
- 1) malý zavaděč na začátku MBR specifický pro OS
- 2) tabulka partition
- - obsah MBR je načten do adresy 0x7c00 a skočí na začátek kódu v MBR (zavaděč)

- - bootujeme

Princip
o Po zapnutí jsou všechny procesory v reálném režimu
o BIOS vybere BSP a ostatní procesory zastaví
o Kód SMP jádra běžící na BSP prohledá paměť na _MP_
o Pokud nenašel, zavede se jednoprocesorové jádro
o Pokud našel, inicializuje APIC BSP
• K tomu je nutné se přepnout do chráněného režimu
o Kód vykonávaný BSP postupně vzbudí AP pomocí Init-IPI (Inter-Processor Interrupt)
o AP se přepne do chráněného režimu a začne svoji další činnost synchronizovat s kódem, který ho spustil a běží na BSP
o Jakmile jsou inicializovány všechny AP, BSP přepne I/O APIC do symetrického IO režimu
• Routovací tabulka, která přesměruje přerušení od sběrnic periferií na některý lokální APIC AP
o SMP jádro pokračuje dál s vlastní inicializací
Viz přednáška PPR d_multithreading.pdf.

http://www.cheesecake.org/sac/smp.html

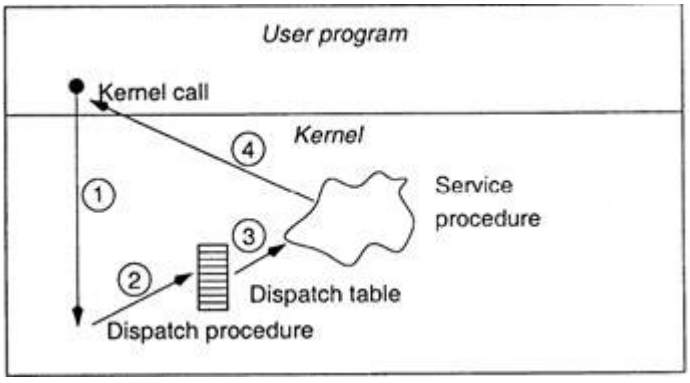
Struktura OS

Monolitické systémy

- pro jednotlivé funkce jsou definovány moduly
- modul může volat jakýkoli jiný modul
- všechny moduly jsou spojeny do vykonatelného souboru s operačním systémem

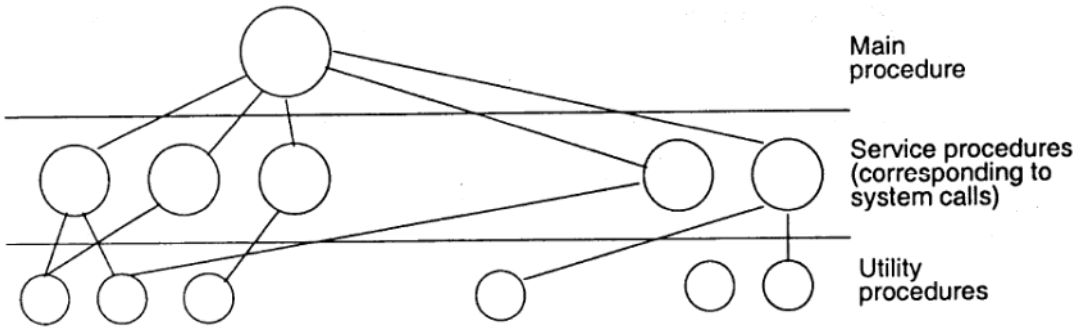
Systémové volání (služba jádra)

- volání vstupního bodu jádra OS s přepnutím do privilegovaného režimu
- zjištění čísla požadované služby
- volání obslužné procedury
- návrat s přepnutím do neprivilegovaného režimu



Model struktury monolitického systému

- hlavní program, který spouští obslužnou proceduru
- množina obslužných procedur pro systémová volání
- podpůrné procedury pro vykonání obslužných procedur
-
-



- mají tendenci extrémně narůstat
- Monolit akumuluje moduly, které by potenciálně mohli být potřebné
- Těžce se ladí

Např. Linux, Windows

Systémy založené na mikrojádře

- **Vrstvené systémy**
  - Hierarchie vrstev poskytujících služby
  - Programy vyšší vrstvy využívají služeb nižších vrstev
  - Holý počítač je nejnižší vrstva
  - Aplikační program je nejvyšší vrstva
  - Princip vrstev umožňuje systematickou tvorbu programů a jejich testování
  - Princip vrstev je možné použít pro monolitický model i pro systémy

- **Mikrojádro**
  - Vrstva nad holým strojem, která obsahuje minimální množinu abstrakcí, tak aby ostatní funkce OS mohly být implementovány nad ním
  - Tyto funkce OS nemusí být vykonávány v privilegovaném režimu
  - Jenom mikrojádro musí být vykonáváno v privilegovaném režimu
  - Typická množina abstrakcí implementována mikrojádrem:
    - Přerušení
    - Vlákna
    - Správa paměti
    - Meziprocesová komunikace
    - Procesy
  - Ostatní funkce – soubory, adresáře, síťové služby – jsou programy vykonávané v uživatelském režimu

základní rozdělení

- monolitické systémy - hlavní program, obslužné procedury, podpůrné procedury
- vrstvené systémy - hierarchie vrstev, nejnižší je holý počítač, nejvyšší je aplikační program

funkční hierarchie - někdy je problém rozdělit do vrstev podle úrovně abstrakce, proto dělení do vrstev podle funkčnosti

- klient-server - obsahuje mikrojádro, které poskytuje pouze základní funkce, většinu práce dělají servery, které jsou oddělené od jádra
- objektově orientovaná struktura - jádro spravuje řadu objektů (zastupují soubory, HW zařízení, ...), mezi objekty jsou tzv. capability = odkaz na objekt + množina práv definujících operace

http://wiki.zvesela.cz/index.php/Funkce_opera%C4%8Dn%C3%ADho_syst%C3%A9mu%2C_struktura_a_rozhran%C3%AD_opera%C4%8Dn%C3%ADho_syst%C3%A9mu%2C_mikroj%C3%A1dro.

Preface

This tutorial is intended as a supplement to the SigOps OS Tutorial to teach the fundamentals of symmetric multprocessing using Intel MP compliant hardware. Knowledge of the concepts and implementations of basic operating system parts such as managing virtual memory and multitasking are assumed and will not be discussed except as they relate to multiprocessing. Knowledge equivalent to an intermediate or advanced computer architecture college course will be helpful in understanding scheduling issues, but is not required.
This tutorial is not intended to be a complete explanation of how to implement an SMP-capable operating system, nor as a replacement for Intel's documentation. Rather it is designed to give an overview of the things I learned in writing SMP support for OpenBLT, a freely redistributable microkernel-based operating system under the BSD licence. Particularly, some tedious hardware aspects will not be discussed in detail when the reader could just as easily read official Intel documentation. The interested reader should refer to the references for more detailed information. For code examples, the reader should refer to the source code of OpenBLT or FreeBSD. The Linux kernel source code might be helpful, although it is under the GPL.
This tutorial is a work in progress. If you see an error or something that needs clarification, please e-mail me.

Terminology

AP
: application processor. A processor that is not the BSP. All APs are in a halted state when the BIOS first gives control to the operating system.

APIC
: Advanced Programmable Interrupt Controller. Either a local APIC or an I/O APIC. It is attached to the APIC bus.

APIC bus
: A special non-architectural bus on which the APICs in the system send messages.

BSP
: bootstrap processor. The processor which is given control after the BIOS finishes its POST.

I/O APIC
: A special APIC for receiving and distributing interrupts from external devices which is backward compatible with the PIC. There is generally only one per computer.

IPI
: interprocessor interrupt. A special interrupt sent to a processor by the originating processor programming its APIC with a target or logical target ID, and an interrupt vector.

Local APIC
: an APIC built in to the processor. It is responsible for dispatching interrupts sent over the APIC bus to its processor core and sending interrupts to other processors over the APIC bus.

MP
: Intel's MultiProcessor Specification, a standard which defines how SMP hardware should be presented to the operating system and how the operating system should interact with this hardware.

serialisation
: The act of executing a certain instruction which causes the processor to pause to retire all instructions currently being executed before proceeding to the next instruction in the stream. For example, before switching to protected mode, the processor must retire all instructions that began executing in real mode before beginning any in protected mode.

SMP
: symmetric multiprocessing. Using multiple processors which share the same physical memory in the same computer at the time. You are probably reading this tutorial with the hope that your operating system will become SMP-capable.

UP
: uniprocessor. Your operating system to date is a UP operating system.

MP Detection and Configuration

When the system first starts, the BIOS detects the hardware installed in the system using electric means and then creates structures to describe this hardware to the operating system. There are two such tables. The first is the MP Floating Pointer Structure, which is required. The second is the MP Configuration Table, which is optional. If the configuration table does not exist, the operating system should set up the default configuration indicated in the floating pointer structure. Some data in the tables is in ASCII. Strings are padded with spaces and are not null-terminated.

First, you need to find the floating pointer structure. According to the spec, it can be in one of four places: (1) in the first kilobyte of the extended BIOS data area, (2) the last kilobyte of base memory, (3) the top of physical memory, or (4) the BIOS read-only memory space between 0xe0000 and 0xfffff. You need to search these areas for the four-byte signature "_MP_" which denotes the start of the floating pointer structure. Absence of this structure indicates that the system is not MP compliant. At this point your operating system can either halt, or it can fall back into a UP setup.

You should checksum the structure to make sure it has not been corrupted. There is not much of interest in the floating pointer structure, unless your system does not have a configuration table. In this case, you will need to get the number of the default configuration your system adheres to and set up the system for SMP using those parameters. Otherwise, you will need to get the address of the configuration table and begin parsing that.

The configuration table is divided into three parts: a header, a base section, and an extended section. The header begins with the four-byte signature "PCMP", although you do not have to search for it. Once you find it, checksum it. At this point, you can print the OEM and product ID strings in the configuration table if you want. You should get the address of the local APIC from this and store it. Then, proceed to parse the base section.

The base section consists of a set of entries that describe either processors, system busses, I/O APICs, I/O interrupt assignments, or local interrupt assignments. All entries are eight bytes in length,

save processor entries which are twenty bytes. The first byte of each entry denotes the type of the entry. Look through each entry. You will probably want to generate quite a few OS-specific data structures here. In particular, you will want to note the APIC ID of each processor in the system, its version, and its type as well as the address of the system's I/O APIC.

Using Local APICs

MP systems have a special bus to which all APICs in the system are connected. This bus is one of the ways the processors can communicate with one another (the other, of course, is shared memory). APICs (both local and I/O) are memory mapped devices. The default location for the local APIC is at 0xfee00000 in physical memory. The local APIC will appear in the same place for each processor, but each processor will reference its own APIC; the APIC intercepts memory references to its registers, and those references will not generate bus cycles on some systems. Since APICs are mapped in high memory, the APs will have to switch to protected mode before they can intialise their local APICs. If you like, you can map the APIC to a different address using the paging unit, but be sure to disable caching in the page table entry since some registers can change between accesses. For this reason, pointers to APIC registers should be volatile. To initialise the BSP's local APIC, set the enable bit in the spurious interrupt vector register and set the error interrupt vector in the local vector table.

Booting Application Processors

Once you have detected the processors in the system, set up your local APIC, and verified that you can communicate with it (hint: read the APIC version register), it's time to boot the APs. N.B. that it is good practise to not try to boot the BSP here. That would be bad.

Since the APs will wake up in real mode, everything they need to get started should be in low memory (below 0x100000 or one megabyte). First, set the shutdown code by setting address 0:f to 0xa. Then, grab a page of memory for the AP's stack. You will also need space to store the `trampoline' code, i.e. the code the processor executes after waking up to switch to protected mode and jump to the kernel. You can either use the same page of code for each processor or store the code at the bottom of the processor's stack. Note that the start of the code must be at a page-aligned address. Copy the code there, then set the warm reset vector at address 40:67 to the start of this code. Next, you should reset a bit in the kernel which the processor will use to signal that it has booted and finished initialisation and clear any APIC error by writing a zero to the error status register. If you need to pass any parameters or data to the AP, now would be a good time to set that up. For example, since OpenBLT's kernel runs in high memory, I have to pass the address of the page directory in memory so that the AP can load it and enable paging before calling the kernel.

Now you can actually boot the processor. The procedure consists of sending a sequence of interrupts to the processor. The incremental effect of each is undefined, but at the end of the sequence, the processor will be booted. First send an INIT IPI. Assert the INIT signal by writing the target processor's APIC ID to the high word of the interrupt command register. Then write to the low word with the bits set to enable the INIT delivery mode, level triggered, and assert the interrupt. Deassert INIT by repeating the procedure with the assert bit reset. Now, wait 10 ms. Use of the APIC timer is suggested.

If the local APIC is not an 82489dx, you need to send two STARTUP IPIs. Clear APIC errors, set the target APIC ID in the ICR, then send the interrupt by writing to the low word of the ICR with bits set for STARTUP delivery mode and with the code vector in the low byte. The code vector is the physical page number at which the processor should start executing, i.e. the start of your trampoline code. Wait 200 ms, then check the low word of the ICR to make sure bit 12 is reset to indicate the interrupt was dispatched before sending the second STARTUP. After sending it, spin and wait for the AP to set its ready bit in memory. You may want to set a timeout of 5 seconds, after which you assume the processor did not wake up.

Switching from Real Mode to Protected Mode

Provided you did everything right above, the processor at some point woke up in real mode and started executing the code you told it to. First, execute a "cli" instruction to turn off interrupts, just in case. Now, begin the switch to protected mode. Load an appropriate value into GDTR. This can either point to the actual GDT or in my case, a temporary GDT. If you need to activate paging, load the address of a page directory into cr3. Then set bit zero in cr0 to enable protected mode as well as bit 31 if you need to enable paging to get into the kernel. Then do an ljmp to the kernel text segment with an offset that points to the next instruction to serialise the processor. Now that you're in protected mode, load appropriate descriptors into the segment registers, then execute a "cld",

which is reportedly what gcc expects. Then, jump to the starting address of your kernel.
Don't reference any symbols in this code since it will be running at an address for which it was not linked; all memory references must be absolute. Since your kernel is above one megabyte in memory, you can't access any global variables in real mode. Also be careful in specifying your offset address for the ljmp instruction, and do specify the address of the start of your kernel, not a symbol in the instruction that goes into the kernel. Jumping to a symbol doesn't seem to work. For details, see OpenBLT's kernel/trampoline.S.
Debugging this part is really not too bad. What you have to do is establish some communication space in low memory, then have the AP write bytes to that memory to explain what it is doing and print these out on the BSP.

Interprocessor Interrupts

IPIs are used to maintain synchronisation between the processors. For example, if a kernel page table entry changes, both processors must either flush their TLBs or invalidate that particular page table entry. Whichever processor changed the mapping knows to do this automatically, but the other processor does not; therefore, the processor which changed the mapping must send an IPI to the other processor to tell it to flush its TLB or invalidate the page table entry.
Using the local APIC, you can send interrupts to all processors, all processors but the one sending the interrupt, or a specific processor or logical address as well as self-interrupts. To send an IPI, write the destination APIC ID, if needed, into the high word of the ICR, then write the low word of ICR with the destination shorthand and interrupt vector set to send the IPI. Be sure to wrap these functions in spinlocks. You might want to turn off interrupts as well while sending IPIs.

Other Considerations

One thing to note is that semaphores (a.k.a. spinlocks) may need to be done differently under SMP. Consider a scenario where semaphores are procured with a ``bts'' instruction. If both processors hit that instruction at the same time while the semaphore is reset, they might both think they have acquired it. For this reason, you would need to use a ``lock'' prefix on that instruction to lock the system bus and maintain synchronisation.

From <http://www.cheesecake.org/sac/smp.html>

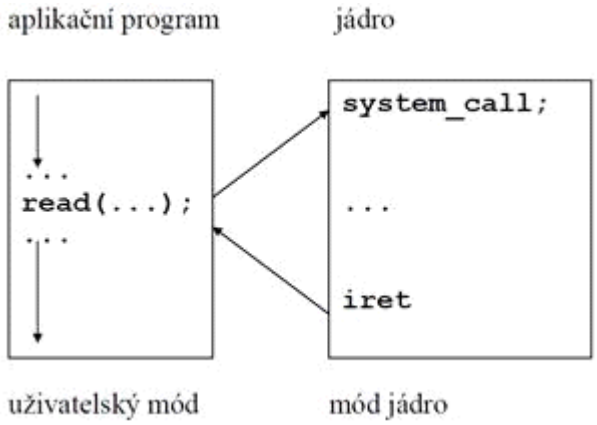
From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Jádro operačního systému - monolitické, hybridní a mikrojádro.

Thursday, May 30, 2013 8:25 AM

Jádro operačního systému

Jádro je speciální program zavedený do hlavní paměti při startu systému přímo vykonávaný HW. Jádro má přístup k adresovému prostoru procesů. Jádro je reentrantní, každý proces má svůj zásobník jádra, často přímo v adresovém prostoru procesu – chráněný, spravovaný jádrem. (procesy se navzájem neovlivňujou)

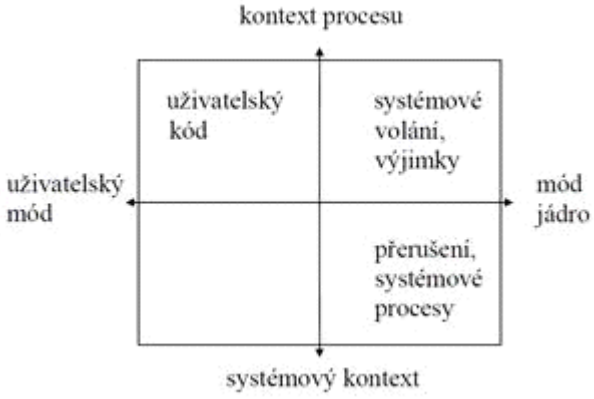


Jádro

- Vykonává služby
- Zpracovává výjimky
- Zpracovává přerušení od periferních zařízení
- Vykonává systémové procesy (správa paměti, přepočítávání priorit procesů)

Jádro pracuje

- V kontextu procesu
- V systémovém kontextu



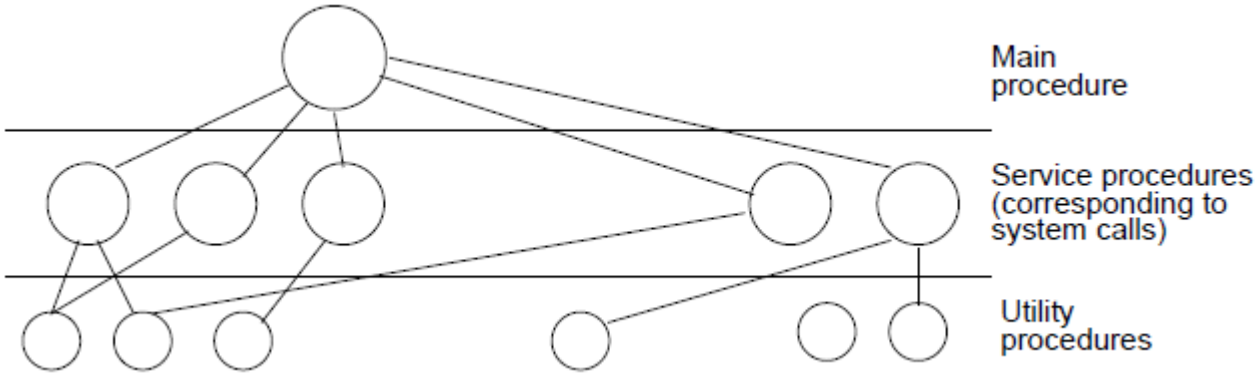
Kontextem procesu rozumíme jeho stav, tj. množinu informací nutnou k obnovení činnosti procesu poté, co byl přerušen.
Je-li OS vykonáván jistý proces, říkáme o něm, že běží v kontextu procesu. Rozhodne-li se OS vykonávat jiný proces, provede **přepnutí kontextu**. OS dovoluje přepnout kontext jen za určitých předpokladů. Při přepínání kontextu si jádro uchovává dostatek informací k pozdějšímu obnovení činnosti procesu.
Obdobně při přechodu z uživatelského režimu do režimu jádra (zde jde o změnu režimu, nikoli změnu kontextu) si jádro uchová dostatek informací k návratu do uživatelského režimu.
Přerušení je obsluhováno v kontextu přerušeného procesu (pro obsluhu přerušení není vytvářen zvláštní proces), ačkoli tento proces nemusel přerušení způsobit.
Kontext procesu obecně
http://www.linfo.org/context_switch.html

Monolitické

- Jeden spustitelný soubor
- Uvnitř moduly pro jednotlivé funkce (filesys, procesy)
- Jeden program, řízení se předává voláním podprogramů
- Horší na údržbu
- Příklady: UNIX, Linux, MS DOS

Typickou součástí jádra je např. souborový systém

Linux je monolitické jádro OS, s podporou zavádění modulů za běhu systému



http://www.512.cz/index.php?title=Rozd%C4%9Blen%C3%AD_OS,_architektura_a_komponenty_OS._Z%C3%A1kladn%C3%AD_funkce_OS

Mikrojádro

- Model klient – server
- Většinu činností OS vykonávají samostatné procesy mimo jádro (servery, např. systém souborů)
- Poskytuje pouze nejdůležitější nízkoúrovňové funkce
    - Nízkoúrovňová správa procesů
    - Adresový prostor, komunikace mezi adresovými prostory
    - Někdy obsluha přerušení, vstupy/výstupy
- Pouze mikrojádro běží v privilegovaném režimu
    - Méně pádů systému

Výhody

- vynucuje modulární strukturu
- Snadnější tvorba distribuovaných OS (komunikace přes síť)

Nevýhody

- Složitější návrh systému
- Režie

Příklady: QNX, **Hurd**, OSF/1, MINIX, Amoeba

Hybridní

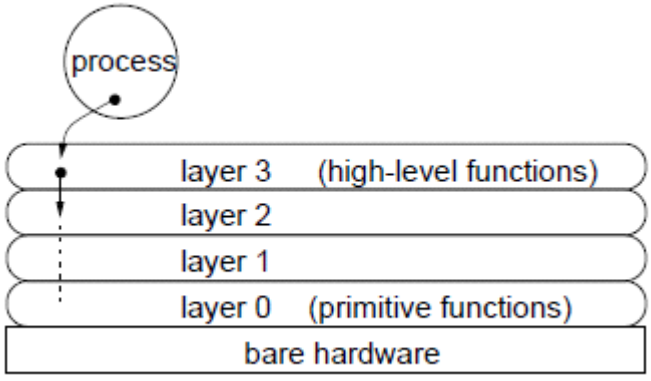
Hybridní jádro je v informatice označení pro jádro operačního systému, které kombinuje vlastnosti monolitického jádra a mikrojádra za účelem získání výhod obou vyhraněných řešení. Hybridní jádro je podobné mikrojádru, ale má některé vlastnosti monolitického jádra, kvůli vyššímu výkonu. Na rozdíl od monolitického jádra nedokáže hybridní jádro za běhu samo zavádět moduly. V jaderném prostoru hybridního jádra běží některé služby (např. implementace síťového protokolu nebo souborový systém), aby se dosáhlo nižší režie v porovnání s mikrojádry, ostatní kód jádra (ovladače zařízení), běží v uživatelském prostoru a označují se jako servery.

- WinNT

http://cs.wikipedia.org/wiki/Hybridn%C3%AD_j%C3%A1dro

Vrstvené

Výstavba systému od nejnižších vrstev Vyšší vrstvy využívají primitiv poskytovaných nižšími vrstvami Hierarchie procesů
–Nejníže vrstvy komunikující s HW
–Každá vyšší úroveň poskytuje abstraktnější virtuální stroj
–Může být s HW podporou – pak nelze vrstvy obcházet (obdoba systémového volání) Příklady: THE, MULTICS



Zásobník jádra:

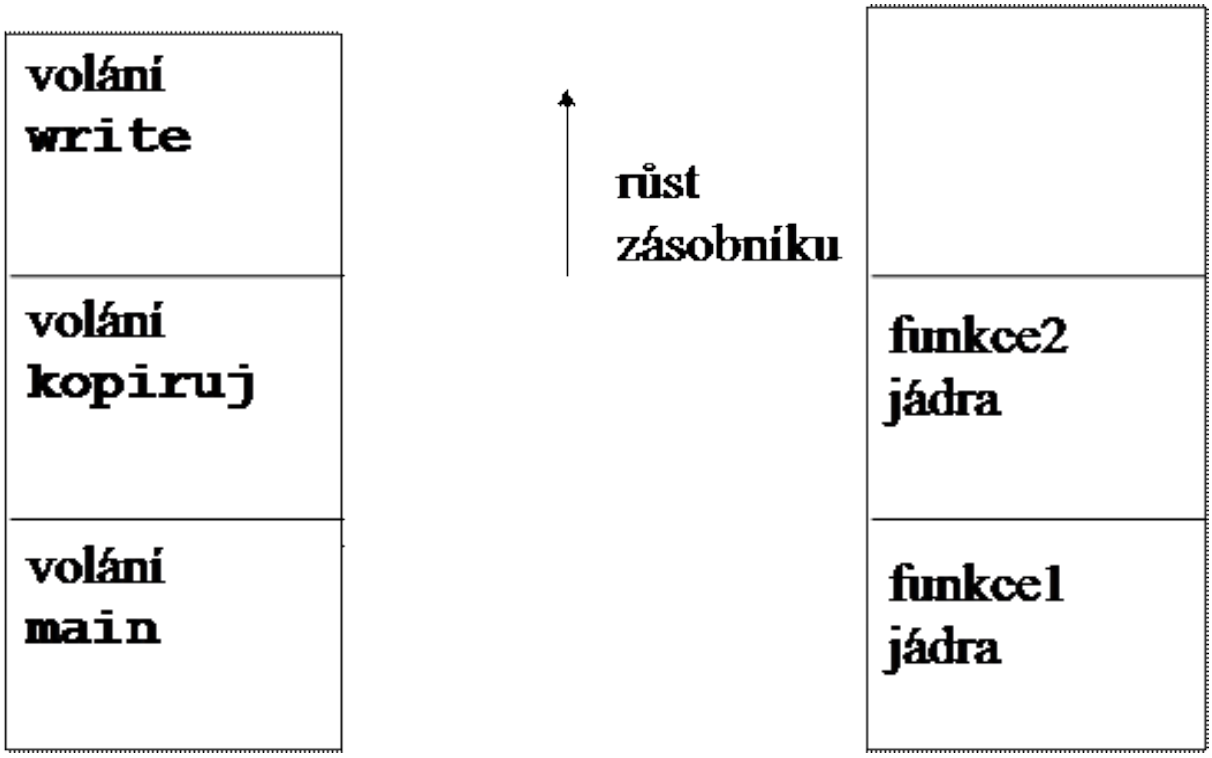
proces v uživatelském módu má přístup ke svému adresovému prostoru, k systémovému prostoru voláním sytem_call()

jádro má přístup k adresovému prostoru procesů
proces používá dva zásobníky

- uživatelský
- jádra

uživatelský zásobník

zásobník jádra



jádro je reentrantní

každý proces má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem

každý proces má položku v tabulce procesů **`proc`** záznam a u (*user*) oblast

u oblast – údaje potřebné když je proces vykonávaný

- tabulku deskriptorů souborů otevřených souborů
- okamžitý adresář
- kořenový adresář
- často zásobník jádra procesu

Vloženo z <http://www.kiv.zcu.cz/~safarikj/vyuka/os/prednasky/prednaska04.doc>

Vloženo z <http://www.kiv.zcu.cz/~safarikj/vyuka/os/prednasky/prednaska04.doc>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Virtuální adresový prostor.

Thursday, May 30, 2013 8:26 AM

ZOS

Viz přednáška 09_2012 ZOS. Program větší než dostupná fyzická paměť => mechanismus překrývání (overlays). Jako řešení tohoto problému se dnes nejčastěji používá právě virtuální paměť.

Překrývání (overlays)

Program – rozdělen na moduly tak, aby se daly postupně zavádět jednotlivé části do paměti, která je menší než celková paměť potřebná pro běh aplikace. (analogie s koberci)

Start – spuštěna část 0, při skončení zavede část 1…

Časté zavádění některých modulů

- Více překryvných modulů + data v paměti současně
- Moduly zaváděny dle potřeby (nejen 0, 1, 2…)
- Mechanismus odkládání (jako odkládání procesů)

Zavádění modulů zařizuje OS

Rozdělení programů i dat na části – navrhuje programátor (Např. vytváření DLL)

- Vliv rozdělení na výkonnost, komplikované
- Pro každou úlohu nové rozdělení

Virtuální paměť

- Potřebujeme rozsáhlý adresový prostor
- Ve skutečné paměti je pouze část adresového prostoru
  - Jinak by to bylo příliš drahé
- Zbytek může být odložen na disku

Virtuální adresy

- Fyzická paměť slouží jako cache virtuálního adresního prostoru procesů
- Proces – používá virtuální adresy
- Pokud požadovaná část VA prostoru je ve fyzické paměti, tak MMU převede VA => FA, přístup k paměti
- Pokud požadovaná část není ve fyzické paměti, OS si ji musí přečíst z disku I/O operace – přidělení CPU jinému procesu
- Většina systémů virtuální paměti používá stránkování

Mechanismus stránkování (paging)

- Program používá virtuální adresy
- Musíme rychle zjistit, zda je požadovaná adresa v paměti – pokud ano, převedeme VA na FA
- Musí být co nejrychlejší, děje se při každém přístupu do paměti

VAP – stránky (pages) pevné délky. Délka je obvykle mocnina 2, nejčastěji se jedná o 4KB, běžně 512B – 8KB

Fyzická paměť – rámce (page frames) stejné délky. **Rámec** může obsahovat **PRÁVĚ JEDNU stránku**. Na **známém místě v paměti** pak musí být **TABULKA STRÁNEK**. Ta poskytuje mapování virtuálních stránek na rámce.

Tabulka stránek

- Součástí PCB (tabulka procesů) – určuje, kde leží jeho tabulka stránek

- Velikost záznamu v tabulce stránek je 32 bitů kde vyšších 20 bitů určuje číslo stránky a nižších 12 bitů určuje offset
- Číslo rámce má pak 20 bitů (takže max. 2^20 stránek)

Výpočet adresy

Velikost stránky = 4096B.

Je dána VA(p1)=100. Určete FA. Tabulka stránek je:

Číslo stránky	Rámec
0	**1**
1	**2**
2	**--**
3	**0**

Máme-li více procesů, každý má svou vlastní tabulku stránek.

Virtuální adresu rozdělíme na číslo stránky a offset.

Str = VA div 4096 (dělení)

Offset = VA mod 4096 (zbytek po dělení)

Převod pomocí tabulky stránek – převedeme číslo stránky na číslo rámce

- tab_str[0] = 1 (pro stránku 0 je číslo rámce 1)
- tab_str[1] = 2
- tab_str[2] = -- stránka není namapována
- tab_str[3] = 0
- Pro VA = 100 je stránka 0, offset 100 => tedy rámec 1

Z čísla rámce a offsetu sestavíme fyzickou adresu:

- FA = rámec * 4096 + offset
- FA = 1 * 4096 + 100
- FA = 4196 v daném případě
- V reálném systému dělení znamená rozdělení na vyšší a nižší bity adresy (proto mocnina dvou)
- Nižší bity – offset
- Vyšší bity – číslo stránky

Výpadek stránky

- stránka není mapována
- Výpadek stránky způsobí výjimku, zachycena OS pomocí přerušení
- OS iniciuje zavádění stránky a přepne na jiný proces
- Po zavedení stránky OS upraví mapování (tabulku stránek)
- Proces může pokračovat
- Pokud daná stránka procesu není namapována na určitý rámec ve fyzické paměti a chceme k ní přistoupit, dojde k výpadku stránky – vyvolání přerušení operačního systému. Operační systém se postará o to, aby danou stránku zavedl do nějakého rámce ve fyzické paměti, nastavil mapování a poté může přístup proběhnout.

- Vnitřní fragmentace (část přidělené paměti je nevyužita), vnější fragmentace (souvislý paměťový prostor mapován do nesouvislých částí paměti)…
- Tabulka stránek procesu – mapuje číslo stránky na číslo fyzického rámce, obsahuje i další informace jako např. příznaky ochrany.
- Relokace = mapování VA na FA
- Ochrana – v tabulce stránek jsou pouze ty stránky, ke kterým má proces přístup. Při přepnutí na jiný proces přepne MMU na jinou tabulku stránek.
- Problémy

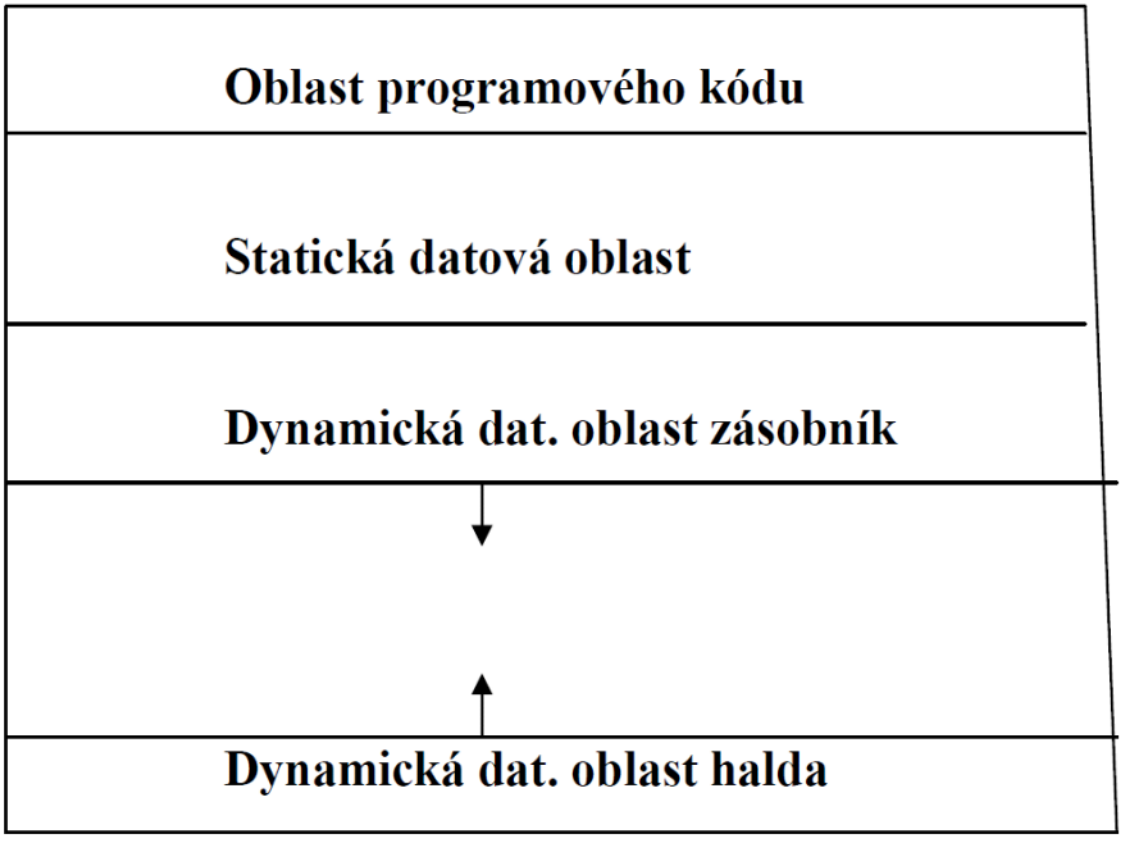
- velikost tabulky stránek – pomůže víceúrovňová struktura
- rychlost převodu VA -> FA – pomůže TLB (Transaction Look-aside Buffer)
- HW cache
- dosáhneme zpomalení jen 5 až 10%
- Přepnutí kontextu na jiný proces – problém (vymazání cache, …); než se TLB opět zaplní – pomalý přístup

Invertovaná tabulka stránek – řešení problému velikosti celé tabulky, obsahuje položky pro každý fyzický rámec. Omezený počet – dán velikostí RAM – VA je 64 bitů, 4KB stránky, 256MB RAM – 65536 položek. Forma položky: (id procesu, číslo stránky).

Při výpadku stránek nastupují algoritmy nahrazování stránek: FIFO, …
OPT, LRU, NRU, second chance, aging, clock

Each active process must have a Page Directory assigned to it. However, there is no need to allocate RAM for all Page Tables of a process at once; it is more efficient to allocate RAM for a Page Table only when the process effectively needs it.
 The physical address of the Page Directory in use is stored in a control register named **cr3** .
 The Directory field within the linear address determines the entry in the Page Directory that points to the proper Page Table. The address's Table field, in turn, determines the entry in the Page Table that contains the physical address of the page frame containing the page. The Offset field determines the relative position within the page frame (see Figure 2-7). Because it is 12 bits long, each page consists of 4096 bytes of data.

Z FJP o rozdělení paměti:



From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Algoritmy nahrazování stránek

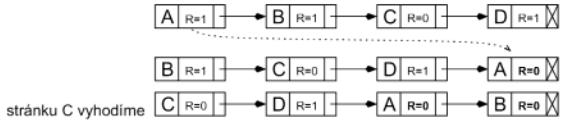
Thursday, May 30, 2013 10:35 AM

```
KIV/ZOS 2003/2004
Přednáška 10

Algoritmus Not-Recently-Used (NRU, NUR)
.......................................

* OS se snaží zjistit, které stránky se používají a nepoužívané vyhazovat
* systémy s VM poskytují HW podporu - stavové bity Referenced (R) a Dirty
  (zde M jako Modified) v tabulce stránek
* bity nastavované HW podle způsobu přístupu ke stránce
 - bit R - nastaven na 1 při čtení nebo zápisu do stránky
 - bit M - nastaven na 1 při zápisu do stránky; označuje, že
           se stránka má při vyhození zapsat na disk
 - po nastavení bitu zůstane na 1 dokud ho SW nenastaví zpět na 0

* algoritmus NRU:
 - na začátku mají všechny stránky nastaveny R=0, M=0
 - bit R je OS nastavován periodicky na 0 (např. při přerušení
   časovače) - tím se rozliší, které stránky byly referencovány
   v poslední době
 - OS rozlišuje 4 kategorie stránek:

   třída 0: R=0, M=0
   třída 1: R=0, M=1  ;; vznikne z třídy 3 po tiku, který nastaví R=0
   třída 2: R=1, M=0
   třída 3: R=1, M=1

 - algoritmus NRU vyhodí stránku z nejnižší neprázdné třídy, výběr mezi
   stránkami ve stejné třídě je náhodný

* algoritmus předpokládá, že je lepší vyhodit modifikovanou stránku
  která nebyla použita 1 tik než nemodifikovanou stránku, která se
  právě používá

* výhody algoritmu NRU:
 - jednoduchost, srozumitelnost
 - efektivně implementovatelný

* nevýhody:
 - výkonnost (jsou i lepší algoritmy)

Pokud by HW neměl bity R a M, můžeme je simulovat následujícím způsobem:

* při startu procesu se všechny jeho stránky označí jako nepřítomné
  v paměti
* při odkazu na stránku nastane výpadek stránky - OS interně nastaví
  R=1 a nastaví mapování v režimu READ ONLY
* při pokusu o zápis do stránky nastane výjimka - OS výjimku zachytí, nastaví
  M=1 a změní režim přístupu do stránky na READ/WRITE.

Algoritmy "Second Chance" a "Clock"
...................................

* algoritmy "Second Chance" a "Clock" vycházejí z algoritmu FIFO

Obchod: V algoritmu FIFO jsme vyhazovali zboží, které bylo zavedeno před
nejdelší dobou (bez ohledu na to, jestli ho někdo chce nebo ne). V algoritmu
"Second Chance" začneme evidovat, jestli zboží někdo v poslední době koupil
(pokud ano, prohlásíme ho za čerstvě zavedené zboží).

* jak modifikovat FIFO, abychom zabránili vyhození často používané stránky?
* algoritmus (Second Chance - vyhledání stránky pro vyhození):
 - podívat se na bit R nejstarší stránky
 - pokud R=0, stránka je nejstarší a zároveň nepoužívaná => vyhodíme
 - pokud R=1, nastavíme R na 0 a přesuneme na konec seznamu stránek
   (jako by byla nově zavedena)
```

Příklad:

Stránky uchováváme v seznamu uspořádaném podle času příchodu. V paměti budeme mít např. stránky A, B, C, D (viz obrázek); algoritmus "Second Chance" bude probíhat v následujících krocích:

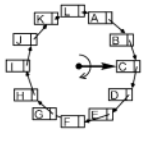
1. krok: Nejstarší je A; má R=1 => nastavíme R na 0 a přesuneme na konec seznamu;
2. krok: Druhá nejstarší je B; má také R=1 => nastavíme R na 0 a opět přesuneme na konec seznamu;
3. krok: Další nejstarší je C, R=0 => vyhodíme jí.



[]

* algoritmus "Second Chance" vyhledává nejstarší stránku, která nebyla referencována "v poslední době"
* pokud byly všechny referencovány, degeneruje na čisté FIFO:
 - postupně všem stránkám nastavíme bit R na 0 a přesuneme je na konec seznamu
 - dostaneme se opět na stránku A, tentokrát má R=0 => vyhodíme jí
 => algoritmus končí nejvýše po $počet_rámců + 1$ krocích

* algoritmus "Clock" = optimalizace datových struktur algoritmu Second Chance:
 - stránky udržovány v kruhovém seznamu
 - ukazatel na nejstarší stránku ("ručička hodin")

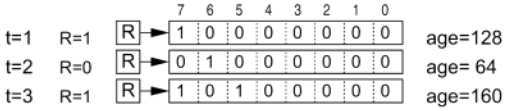


* výpadek stránky -> vyhledáváme stránku k vyhození
 - stránka kam ukazuje "ručička":
  . má-li R=0, stránku vyhodíme a ručičku posuneme o 1 pozici
  . má-li R=1, nastavíme R na 0, ručičku posuneme o 1 pozici; opakujeme dokud nenalezneme stránku s R=0
* od algoritmu Second Chance se liší pouze implementací
* varianty algoritmu Clock používají např. systémy BSD UNIX

Softwarová aproximace LRU
.....

* algoritmus LRU vždy vyhazuje nejdéle nepoužitou stránku

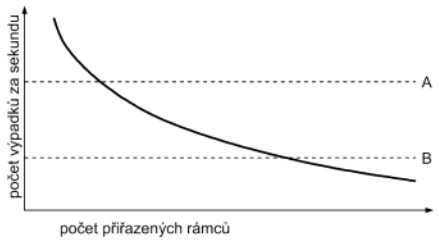
* algoritmus Aging:
 - každá položka v tabulce stránek má pole "stáří" age, N bitů (např. N=8)
 - na počátku age=0
 - při každém přerušení časovače pro každou stránku:
  . posun pole "stáří" o 1 bit vpravo
  . zleva se přidá hodnota bitu R
  . nastavení R na 0



```
* to odpovídá následujícímu kódu (v Turbo Pascalu):

    age := age shr 1;           { posun o 1 bit vpravo }
    age := age or (R shl N-1);  { zleva se přidá hodnota bitu R }
    R := 0;                     { nastavení R na 0 }

* při výpadku stránky se vyhodí stránka, jejíž pole age je má nejnižší
  hodnotu

Dva rozdíly od LRU:

* několik stránek může mít stejnou hodnotu pole age a nevíme která stránka
  byla odkazovaná dříve (u LRU to víme vždy)
 - rozlišení je "hrubé" (= po tricích časovače)
* pole age se může snížit na 0 - nevíme, zda stránka byla naposledy
  odkazovaná před 9 nebo před 1000 tiky časovače
 - uchovává pouze omezenou historii
 - v praxi není problém: pokud je tik časovače po 20 ms a N=8, nebyla
   odkazována 160 ms => nejspíš není tak důležitá, můžeme jí vyhodit
* pokud se musíme rozhodovat mezi dvěma stránkami se stejnou hodnotou
  age, vybíráme náhodně

Shrnutí algoritmů pro nahrazování stránek
.........................................

* optimální algoritmus (MIN čili OPT)
 - není implementovatelný, ale je užitečný pro srovnání
* FIFO
 - vyhazuje nejstarší stránku
 - jednoduchý, ale je chopen vyhodit důležité stránky a trpí Beladyho
   anomálií
* LRU (Least Recently Used)
 - výborný
 - implementace vyžaduje speciální HW, proto prakticky používán zřídka
* NRU (Not Recently Used)
 - rozděluje stránky do 4 kategorií podle bitů R a M
 - efektivita nic moc, přesto občas používán
* "Second chance" a "Clock"
 - vycházejí z FIFO, před vyhozením zkontrolují zda se stránka používala
 - mnohem lepší než FIFO
 - používané algoritmy (např. některé varianty UNIXu)
* Aging
 - dobře aproximuje LRU => efektivní
 - často prakticky používaný algoritmus


Ostatní problémy stránkované virtuální paměti
---------------------------------------------

Alokace fyzických rámců
.......................

* 2 základní metody - globální a lokální alokace:
 - globální alokace - pro vyhození se uvažují všechny rámce
 - lokální alokace - pro vyhození se uvažují pouze rámce alokované
   procesem (tj. obsahující stránky procesu, jehož výpadek stránky se
   obsluhuje)
  . počet stránek alokovaných pro proces se nemění
  . program se vzhledem ke stránkování chová přibližně stejně při
    každém běhu
 - u globální alokace vybírá ze všech rámců
  . lepší průchodnost systému - proto globální alokace častější
  . na běh procesu má vliv chování ostatních procesů

* při lokální alokaci - kolik rámců dát kterému procesu?
 - nejjednodušší - všem procesům dáme stejně, ale potřeby procesů mohou
   být různé
 - proporcionální - dáme každému proporcionální díl podle velikosti procesu
```

- nejlepší metoda - podle frekvence výpadků stránek (Page Fault Frequency, PFF)

Pro většinu rozumných algoritmů se PFF snižuje s množstvím přidělených rámců:



PFF se snažíme udržet v rozumných mezích:

* pokud je PFF větší než A, přidáme procesu rámce
* pokud je PFF menší než B, proces má asi příliš paměti, rámce mu mohou být odebrány

Zahlcení
.....

* proces pro svůj rozumný běh potřebuje pracovní množinu stránek
* pokud se pracovní množiny stránek aktivních procesů nevejdou do paměti, nastane tzv. zahlcení (angl. trashing)

* jak to vypadá:
 - v procesu nastane výpadek stránky
 - paměť je plná (není volný rámec) => je třeba některou stránku vyhodit
 - stránka pro vyhození bude ale brzy zapotřebí, takže se bude muset vyhodit jiná používaná stránka
 - z uživatelského hlediska se to projeví tak, že systém pracuje intenzivně s diskem a běh procesů řádově zpomalí (stráví víc času stránkováním než během)

* řešení - při zahlcení snížit úroveň multiprogramování (zahlcení lze detekovat pomocí PFF)

Zhodnocení mechanismu virtuální paměti
.....

Virtuální paměť má podstatné výhody oproti předchozím mechanismům:

* rozsah virtuální paměti (např. 2 GB pro proces - NT nebo Linux na i386)
 - adresový prostor úlohy není omezen velikostí fyzické paměti
 - multiprogramování (= počet procesů) není zásadně omezeno rozsahem fyzické paměti
* efektivnější využití fyzické paměti
 - není vnější fragmentace paměti
 - nepoužívané části adresového prostoru úlohy nemusejí být ve fyzické paměti

Nevýhody:

* režie při převodu virtuálních adres na fyzické adresy
* režie procesoru (údržba tabulek stránek a rámců, výběr stránek pro vyhození, plánování I/O)
* režie I/O při čtení/zápisu stránky
* paměťový prostor pro tabulky stránek
* vnitřní fragmentace

```
Segmentace
----------

* dosud diskutovaná virtuální paměť byla jednorozměrná:
  od adresy 0 do nějaké maximální virtuální adresy
* pro mnoho programů by bylo výhodnější mít víc samostatných virtuálních
  adresových prostorů
* příklad - mám několik tabulek a chci, aby jejich velikost mohla růst
=> paměť nejlépe mnoho nezávislých adresových prostorů = segmenty

* segment = logické seskupení informací
* každý segment lineární posloupnost adres, začínající od adresy 0
* programátor o segmentech ví, používá explicitně (adresuje konkrétní
  segment)
* příklad - překladač Pascalu může používat samostatné segmenty pro:
  - kód přeloženého programu
  - globální proměnné
  - hromadu
  - zásobník návratových adres
  - je možné i jemnější dělení (segment pro každou proceduru/fci)
* často se používá také pro implementaci:
  - přístupu k souborům (1 soubor = 1 segment)
    . není třeba open, read...
  - sdílených knihoven:
    . dnešní programy využívají rozsáhlé knihovny - knihovnu potřebuje prakticky každý
      program
    . myšlenka vložit knihovnu do segmentu a sdílet mezi více programy

* každý segment je logická entita - má smysl, aby měl samostatnou ochranu

Čistá segmentace
................

* každý odkaz do paměti se skládá z dvojice: (selektor, offset)
 - selektor: číslo segmentu, určuje segment
 - offset: relativní adresa v rámci segmentu
* technické prostředky musí přemapovat dvojici (selektor, offset) na fyzickou
  (= lineární) adresu
* k tomu slouží tabulka segmentů, každá položka tabulky obsahuje:
  - počáteční adresu segmentu (bázi)
  - rozsah segmentu (limit)
  - příznaky ochrany segmentu (nejčastěji čtení, zápis, provádění = rwx)
* postup při převodu na fyzickou adresu:
  - PCB obsahuje odkaz na tabulku segmentů procesu
  - odkaz do paměti má tvar (selektor, offset)
    např. v důsledku instrukce LD R, selektor:offset
  - selektor = index do tabulky segmentů
  - zkontroluje se zda je offset<limit; ne => chyba porušení ochrany paměti
  - zkontroluje se, zda dovolen způsob použití; ne => chyba porušení
    ochrany paměti
  - adresa = báze + offset
* často možnost sdílet segment mezi více procesy

* mnoho věcí podobných jako přidělování paměti po sekcích, ale rozdíl:
  - po sekcích - pro procesy
  - segmenty - pro části procesu
* stejné problémy jako přidělování paměti po sekcích: externí
  fragmentace paměti, mohou zůstávat malé díry (tj. dále již prakticky
  nepoužitelné)

Segmentace na žádost
....................

* segment může být zavedený v paměti nebo odložený na disku
* pokus o adresování segmentu, který není v paměti způsobí výpadek segmentu
* OS zavede segment do paměti
* není-li místo, je některý jiný segment odložen na disk

* HW podpora - v tabulce segmentů bity:
```

Obsluha přerušení, výjimek a systémových volání.

Thursday, May 30, 2013 8:26 AM

Zdroj: http://www.kiv.zcu.cz/~safariki/vyuka/os/prednasky/prednaska05.pdf

http://stackoverflow.com/questions/9968028/returning-from-kernel-mode-to-user-mode

Přerušení

metoda pro asynchronní obsluhu událostí, kdy procesor přeruší vykonávání sledu instrukcí, vykoná obsluhu přerušení a pak pokračuje.

- **vnitřní** - vyvolané procesorem (problém se zpracováním instrukcí, dělení 0, výpadek stránky)
  - Také nazývané **synchronní**, a to proto, že CPU jej vyvolá až po dokončení aktuálně vykonávané instrukce, Intelovský manuál je nazývá **Exception**
- **vnější** - hardwarové, přichází z V/V zařízení, mají přidělena čísla IRQ (Interrupt Request)
  - pro signalizaci přerušení– *kanály přerušení*, reprezentovány čísly IRQ, na jednom IRQ kanálu může být napojeno více zařízení (= sdílený kanál, sdílené IRQ)
  - Také nazývané **asynchronní**, a to proto, že závisí na ticích časovače a může nastat v době kdy CPU právě vykonává nějakou instrukci, Intelovský manuál je nazývá **Interrupt**
- **softwarové** - speciální instrukce INT 0x80 nebo SYSENTER (SYSCALL)

Další dělení je na maskovatelná a nemaskovatelná

Interrupts:
- **Maskable interrupts**: All Interrupt Requests (IRQs) issued by I/O devices give rise to maskable interrupts . A maskable interrupt can be in two states: masked or unmasked; a masked interrupt is ignored by the control unit as long as it remains masked.
- **Nonmaskable interrupts**: Only a few critical events (such as hardware failures) give rise to nonmaskable interrupts . Nonmaskable interrupts are always recognized by the CPU.

- Přerušení mají priority - obsluha přerušení může být přerušena přerušením s vyšší prioritou :) - pozn. Priority z hlediska kernelu jsou jen **low** a **high**, tedy přerušitelné a nepřerušitelné. APIC (Advanced Programmable Interrupt Controller ale priority podporuje, takže priority přerušení jako takové Linux nikde neřeší, ale nechává to na hardware (APIC je v každém CPU).

The Linux kernel is reentrant (like all UNIX ones), which simply means that multiple processes can be executed by the CPU. Hedoesn't have to wait till a disk access read is handled by the deadly slow HDD controller, the CPU can process some other stuff until the disk access is finished (which itself will trigger an interrupt if so). Generally, an interrupt can be interrupted by an other interrupt (preemption), that's called 'Nested Execution'. Depending onthe architecture, there are still some critical functions which have to run without interruption (non-preemptive) by completely disabling interrupts. On x86, these are some time relevant functions (`time.c`, `hpet.c`) and some `xen` stuff.
There are only two priority levels concerning interrupts: 'enable all interrupts' or 'disable all interrupts', so I guess your "high priority interrupt" is the second one. This is the only behavior the Linux kernel knows concerning interrupt priorities and has nothing to do with real-time extensions.
If an interruptible interrupt (your "low priority interrupt") gets interrupted by an other interrupt ("high" or "low"), the kernel saves the old execution code of the interrupted interrupt and starts to process the new interrupt. This "nesting" can happen multiple times and thus can create multiple levels of interrupted interrupts. Afterwards, the kernel reloads the saved code from the old interrupt and tries to finish the old one.

From <http://unix.stackexchange.com/questions/7124/re-entrency-of-interrupts-in-linux>

- Přerušení je obecně asynchronní vzhledem k přerušenému procesu
- Zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící
- čas zpracování přerušení je započítán přerušenému procesu, při zpracování se tedy přistupuje do jeho záznamu proc

Obsluha:

1) uloží IRQ (Interrupt ReQuest) a obsah registrů

2) pošle potvrzení PIC (Programmable Interrupt Controller), který zajišťuje provoz přerušení

3) *modul pro obsluhu přerušení* (Interrupt Handler) vykoná obsluhu přerušení

4) ukončí skokem na ret_from_intr()

Výjimky

- synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)

Zpracování:

- Stejné jako u přerušení, 80x86 procesory mají přibližně 20 definovaných výjimek, každá je zpracována přiřazeným exception handlerem - ten většinou nedělá nic jiného než že pošle signál procesu, který výjimku způsobil, kernel musí poskytnout exception handler pro každou definovanou výjimku, příklady výjimek i s jejich číslem: 0 - Divide Error = dělení nulou, 1 - Debug, 4 - Overflow = přetečení...

1) uloží obsah registrů

2) zpracuje výjimku (funkce v jazyku C)

  - pošle signál procesu
  - zpracuje žádost o stránku

3) ukončí se voláním funkce ret_from_exception()

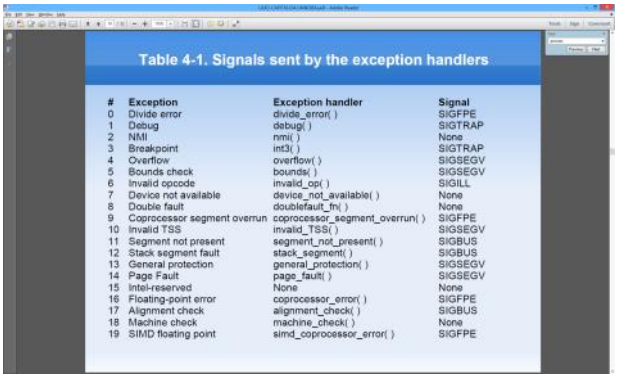


Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
0	Divide error	divide_error( )	SIGFPE
1	Debug	debug( )	SIGTRAP
2	NMI	nmi( )	None
3	Breakpoint	int3( )	SIGTRAP
4	Overflow	overflow( )	SIGSEGV
5	Bounds check	bounds( )	SIGSEGV
6	Invalid opcode	invalid_op( )	SIGILL
7	Device not available	device_not_available( )	None
8	Double fault	doublefault_fn( )	None
9	Coprocessor segment overrun	coprocessor_segment_overrun( )	SIGFPE
10	Invalid TSS	invalid_TSS( )	SIGSEGV
11	Segment not present	segment_not_present( )	SIGBUS
12	Stack segment fault	stack_segment( )	SIGBUS
13	General protection	general_protection( )	SIGSEGV
14	Page Fault	page_fault( )	SIGSEGV
15	Intel-reserved	None	None
16	Floating-point error	coprocessor_error( )	SIGFPE
17	Alignment check	alignment_check( )	SIGBUS
18	Machine check	machine_check( )	None
19	SIMD floating point	simd_coprocessor_error( )	SIGFPE

Vektor přerušení

- 0-31 vnitřní přerušení (výjimky), 32 - 255 IRQ přerušení

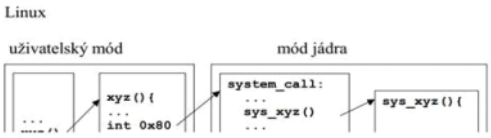
Vektor přerušení u x86 se nazývá IDT (Interrupt Descriptor Table) - tabulka, která přiřazuje obslužnou rutinu ke každému přerušení

Systémové volání

- mechanismus pro volání funkcí operačního systému aplikacemi
- v standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá softwarovým přerušením proceduře jádra syscall()
- system_call - jedna pro všechny služby, požadovaná služba se odlišuje parametrem procedury, který se nazývá číslo

je dulezity registr idtr - obsahuje poiner na tabulku preruseni. prijde preruseni od hardware, dostane se po dratu do i/o apic, ten ho prelozi a ziska cislo preruseni - koukne do tabulky a podle priority v tabulce ho preposle APIC v jadru ktere je uvedeno v te tabulce u toho preruseni. to konkretni jadro dostane preruseni po intrl (interrupt line) a pushne vse co dela na zasobnik, prepne mod do Ring 0 (rezim jadra je jen mod opravneni v CPU, ring 0 znamena ze ti CPU umozni delat cokoli, ring 3 je mod kde bezi aplikace a pamet je omezena jen na tu ktera je pridelena procesu), CPU v tomhle rezimu jadra koukne na IDTR registr kde je adresa vektoru preruseni, pripocte k te adrese 31 (vnitrni preruseni CPU jsou na zacatku vektoru preruseni, pak az jsou ty hardwarovy), koukne kam to ukazuje a dostane pointer na obsluhu preruseni kam taky skoci (porad v rezimu jadra). Ta obsluha muze byt v driveru (interrupt request handler si registruje driver pri zavedeni a je to jen prepsani te adresy ve vektoru preruseni, popr. kdyz ma vic preruseni stejny cislo tak tam je ulozen spojovy seznam a zkousi se ty handlery pekne postupne). obsluha preruseni musi byt co nejkratsi a na konci se musi zavolat instrukce iret, ktera koukne na zasobnik jadra a podle toho co tam uvidi bud vrati rizeni zpet do procesu kterej byl prerusenej, nebo v pripade zanorenyo preruseni vrati obsluhu predchozimu preruseni. Pokud je preruseni !na dlouho!, linux v jeho obsluze jen vyvola nejaky tasklet (lwp proces v jadre) a necha obsluhu na pozdeji (linux checkuje jestli nejake tasklety cekaji a obsluhuje je napr. pri prepinani do/z rezimu jadra, ale takovych mist je v kernelu pry vic.

systémového volání



Obsluha:

system_call:

1) uloží obsah registrů (HW kontext)

2) zavolá odpovídající funkci (v jazyku C)

3) ukončí se voláním ret_from_sys_call()

Int 0x80 je legacy instrukce, funguje jen u 32 bit systému a dnes se nepoužívá

- `syscall` is default way of entering kernel mode on `x86-64`. This instruction is not available in 32 bit modes of operation *on Intel processors.*
- `sysenter` is an instruction most frequently used to invoke system calls in 32 bit modes of operation. It is similar to `syscall`, a bit more difficult to use though, but that is kernel's concern.
- `int 0x80` is a legacy way to invoke a system call and should be avoided.
  Preferable way to invoke a system call is to use VDSO, a part of memory mapped in each process address space that allow to use system calls more efficiently (for example, by not entering kernel mode in some cases at all). VDSO also takes care of more difficult, in comparison to the legacy `int 0x80` way, handling of `syscall` or `sysenter` instructions.

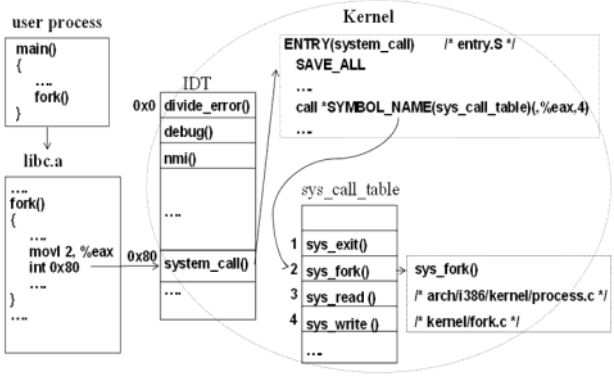
From <http://stackoverflow.com/questions/12806584/what-is-better-int-0x80-or-syscall>

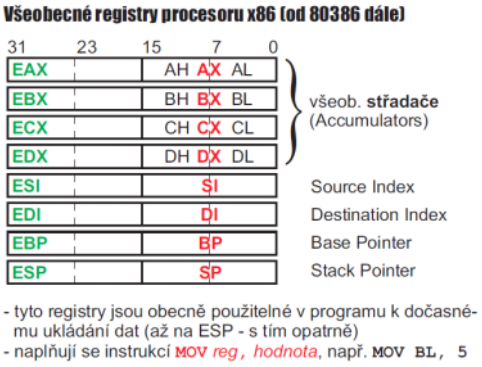
VDSOs (**V**irtual **D**ynamically linked **S**hared **O**bjects) are a way to export kernel space routines to user space applications, using standard mechanisms for linking and loading (i.e. standard ELF format).
It helps to reduce the calling overhead on simple kernel routines, and also can work as a way to select the best system call method on some architectures.
An advantage over other methods is that such exported routines can provide proper DWARF debugging information.
Implementation generally implies hooks in the dynamic linker to find the **VDSO**s.

From <http://en.wikipedia.org/wiki/VDSO>

Cool diagrámek, jak se zpracovává systémové volání:

(z diskuse http://www.unix.com/unix-dummies-questions-answers/178442-x86-interrupts-system-calls.html)



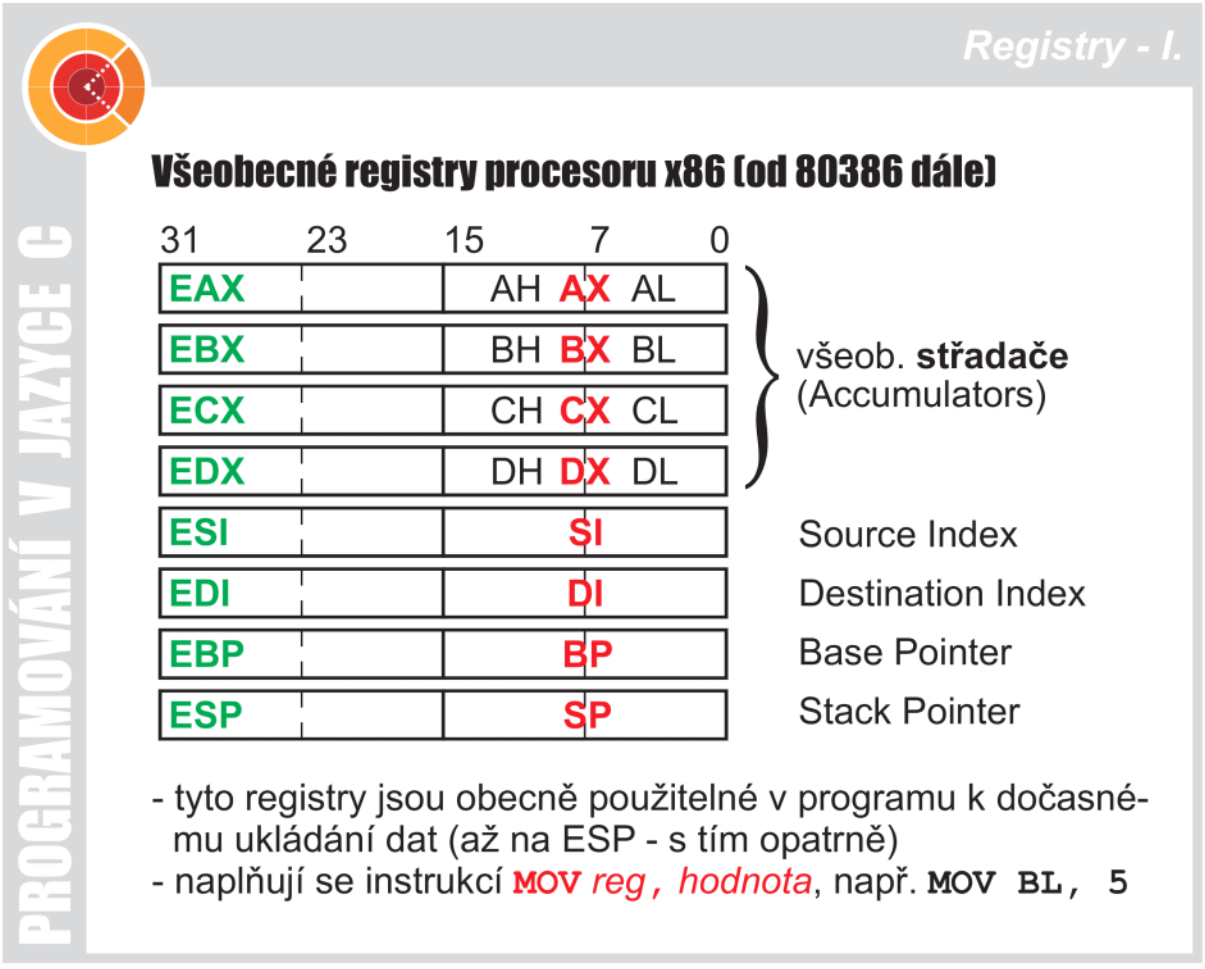


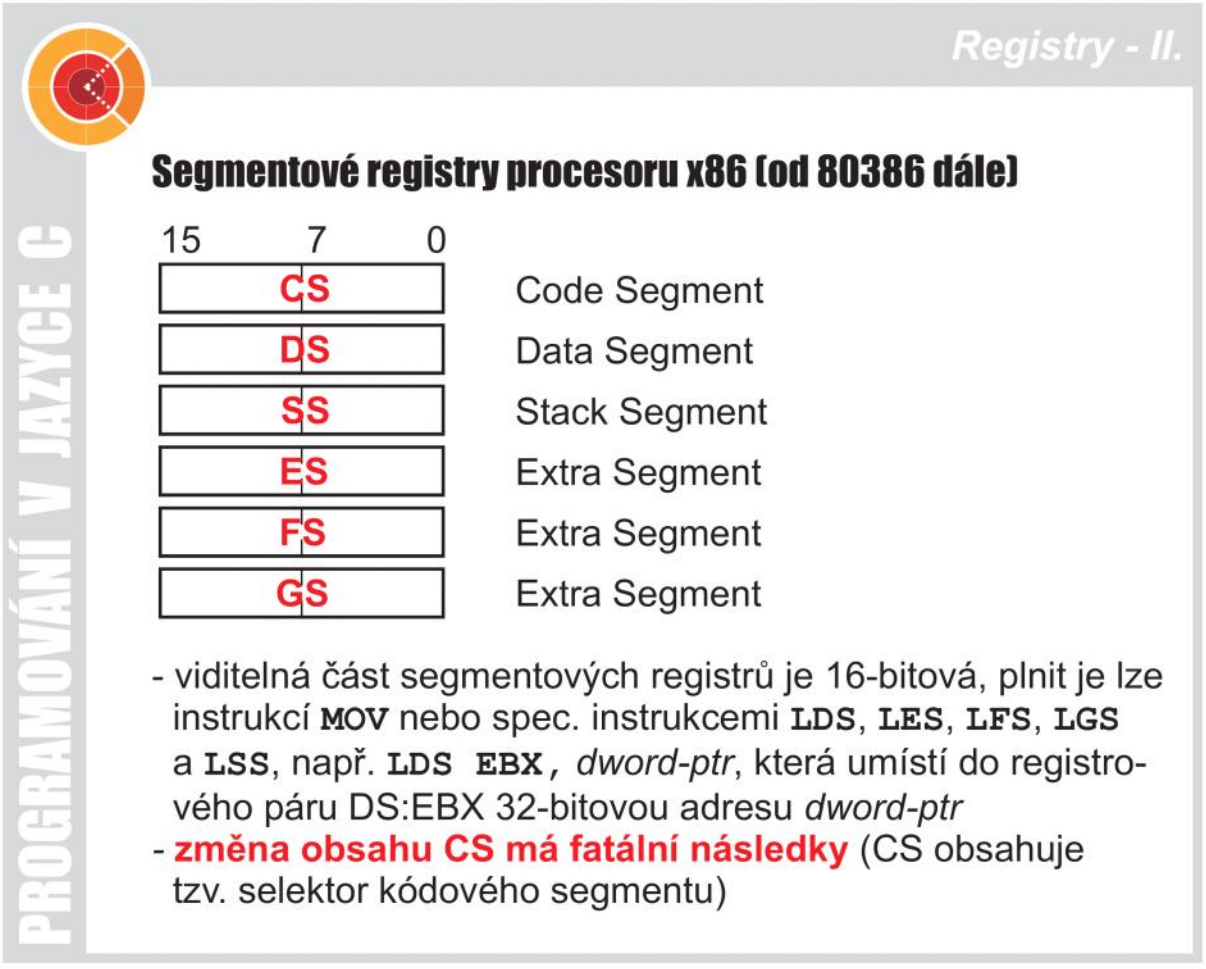
From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

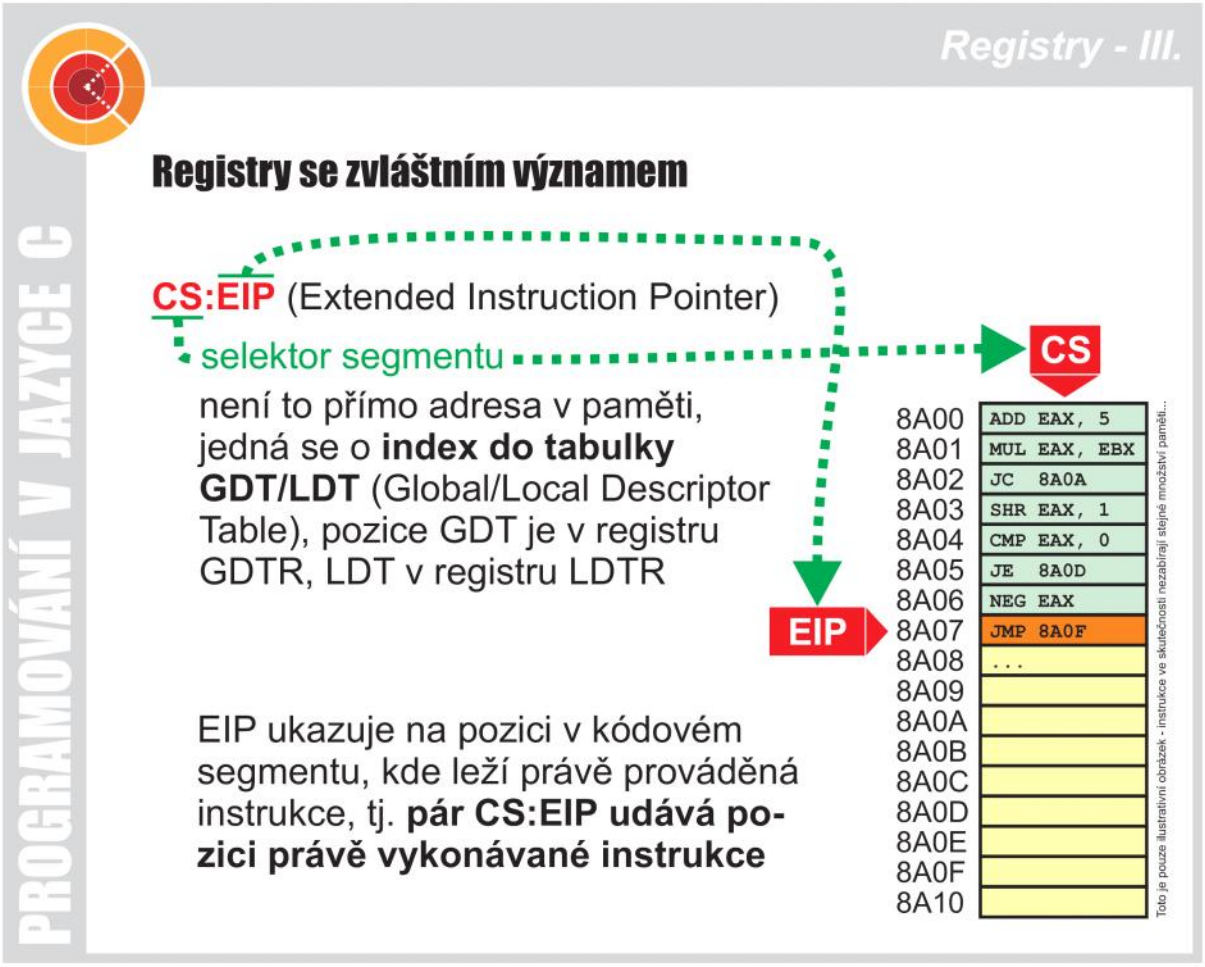
PROGRAMOVÁNÍ V JAZYCE C

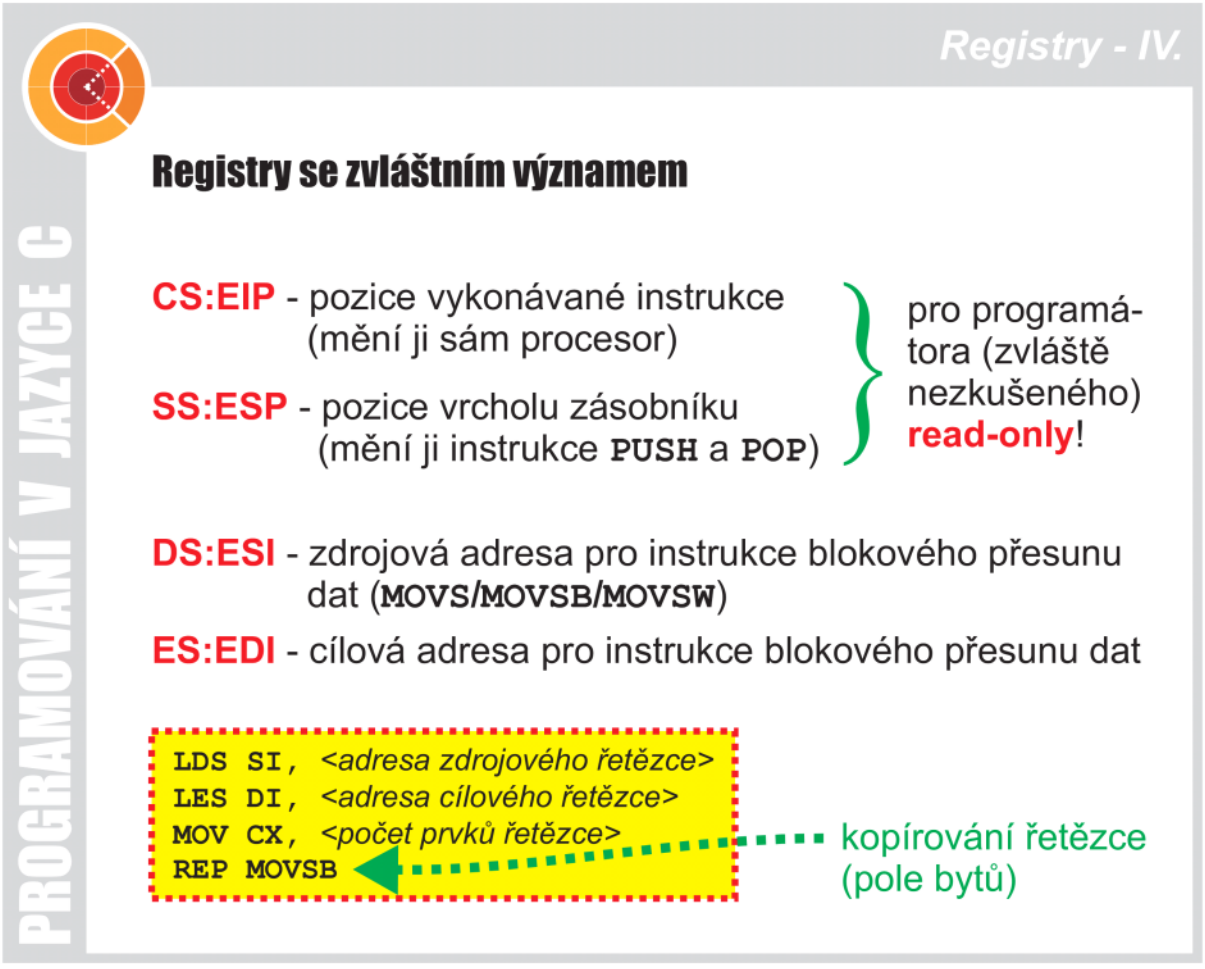
Programátorský model x86

- programátorským modelem se rozumí soubor vlastností a fyzických součástí procesoru, které ovlivňují jeho programování v nízkoúrovňových jazycích
- zejména popisuje **uspořádání paměti**, využitelné **registry**, nativní **typy dat** daného procesoru, **instrukční soubor**, **přerušovací systém**, atp.


Registry - I.
PROGRAMOVÁNÍ V JAZYCE C
Všeobecné registry procesoru x86 (od 80386 dále)
31 23 15 7 0
EAX AH AX AL
EBX BH BX BL
ECX CH CX CL
EDX DH DX DL
všeob. střadače (Accumulators)
ESI SI Source Index
EDI DI Destination Index
EBP BP Base Pointer
ESP SP Stack Pointer
- tyto registry jsou obecně použitelné v programu k dočasnému ukládání dat (až na ESP - s tím opatrně)
- naplňují se instrukcí MOV reg, hodnota, např. MOV BL, 5

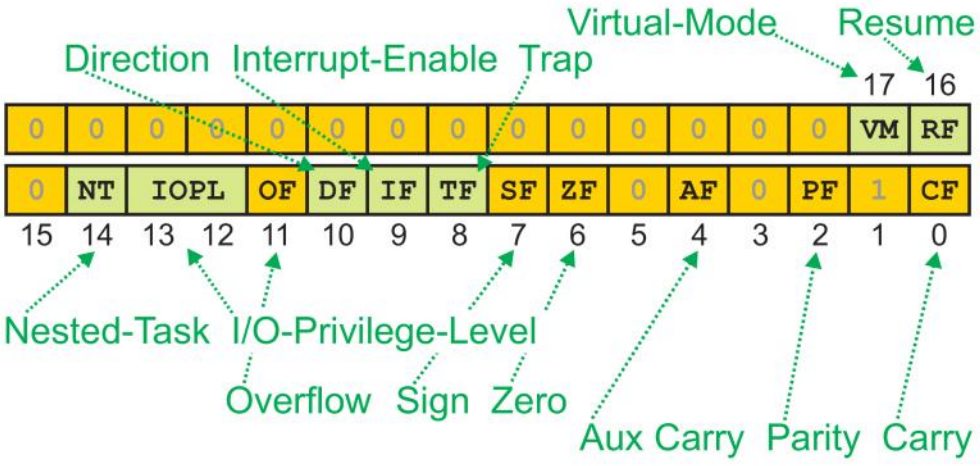

Registry - II.
PROGRAMOVÁNÍ V JAZYCE C
Segmentové registry procesoru x86 (od 80386 dále)
15 7 0
CS
DS
SS
ES
FS
GS
Code Segment
Data Segment
Stack Segment
Extra Segment
Extra Segment
Extra Segment
- viditelná část segmentových registrů je 16-bitová, plnit je lze instrukcí MOV nebo spec. instrukcemi LDS, LES, LFS, LGS a LSS, např. LDS EBX, dword-ptr, která umístí do registrového páru DS:EBX 32-bitovou adresu dword-ptr
- změna obsahu CS má fatální následky (CS obsahuje tzv. selektor kódového segmentu)

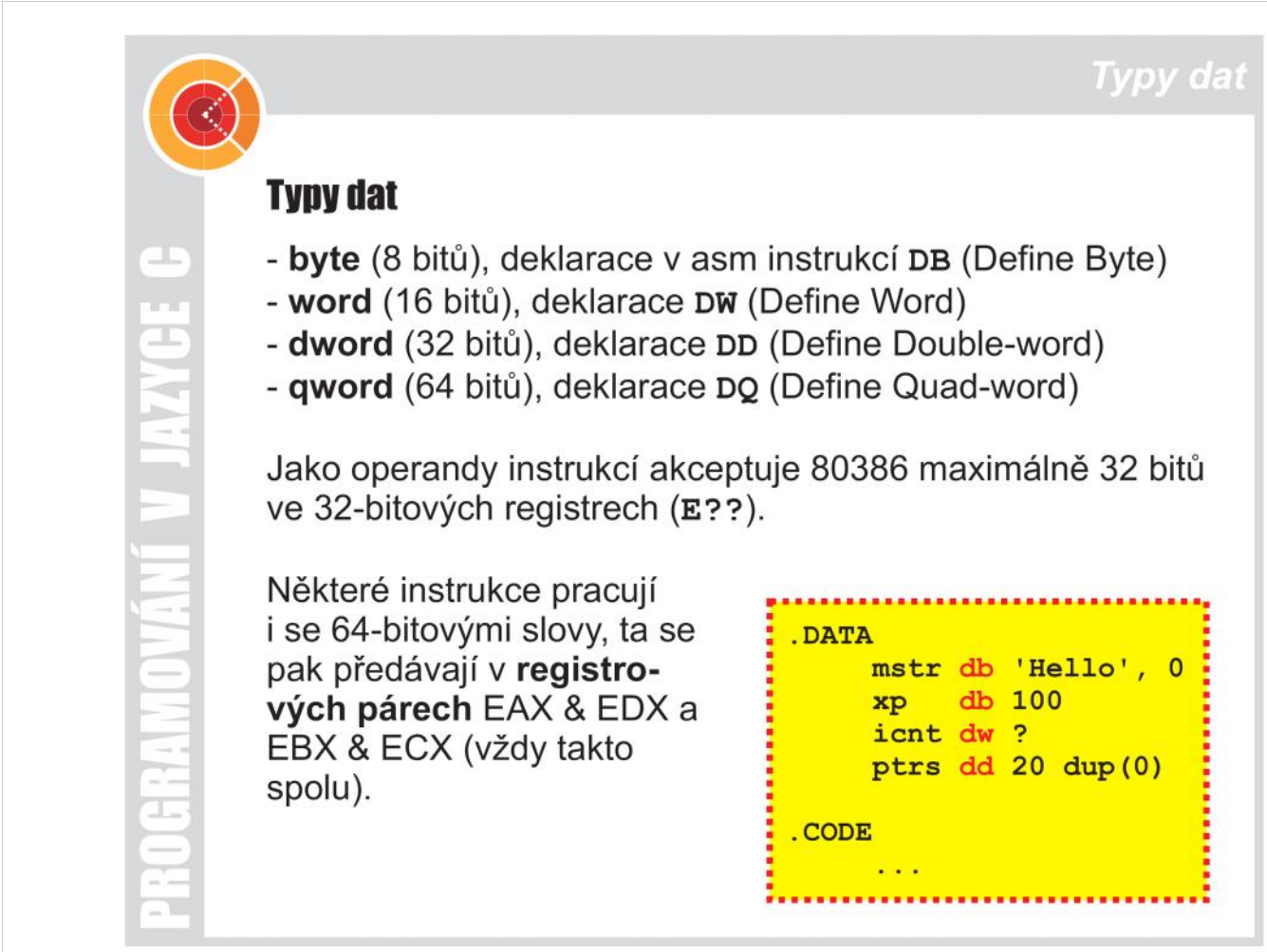

Registry - III.
PROGRAMOVÁNÍ V JAZYCE C
Registry se zvláštním významem
CS:EIP (Extended Instruction Pointer)
selektor segmentu
není to přímo adresa v paměti, jedná se o index do tabulky GDT/LDT (Global/Local Descriptor Table), pozice GDT je v registru GDTR, LDT v registru LDTR
EIP ukazuje na pozici v kódovém segmentu, kde leží právě prováděná instrukce, tj. pár CS:EIP udává pozici právě vykonávané instrukce
CS
EIP
8A00 ADD EAX, 5
8A01 MUL EAX, EBX
8A02 JC 8A0A
8A03 SHR EAX, 1
8A04 CMP EAX, 0
8A05 JE 8A0D
8A06 NEG EAX
8A07 JMP 8A0F
8A08 ...
8A09
8A0A
8A0B
8A0C
8A0D
8A0E
8A0F
8A10
Toto je pouze ilustrativní obrázek - instrukce ve skutečnosti nezabírají stejné množství paměti...


PROGRAMOVÁNÍ V JAZYCE C
Registry - IV.
Registry se zvláštním významem
CS:EIP - pozice vykonávané instrukce (mění ji sám procesor)
SS:ESP - pozice vrcholu zásobníku (mění ji instrukce PUSH a POP)
pro programátora (zvláště nezkušeného) read-only!
DS:ESI - zdrojová adresa pro instrukce blokového přesunu dat (MOVS/MOVSB/MOVSW)
ES:EDI - cílová adresa pro instrukce blokového přesunu dat
LDS SI, <adresa zdrojového řetězce>
LES DI, <adresa cílového řetězce>
MOV CX, <počet prvků řetězce>
REP MOVSB
kopírování řetězce (pole bytů)

Příznakový registr EFLAGS (32-bitový)

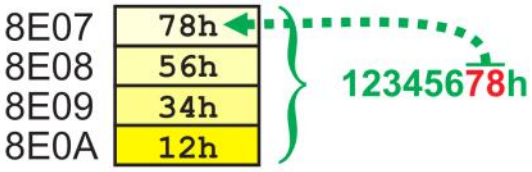
Obsahuje dva druhy **vlajek** (**1-bitových příznaků**):
(i) **nastavované procesorem** po provedení instrukce indikují vlastnosti výsledku (**CF**, **PF**, **AF**, **ZF**, **SF**, **OF**),
(ii) **nastavované programátorem** řídí činnost procesoru (**TF**, **IF**, **DF**, **VM**, **RF**, **NT**, **IOPL**).




Typy dat
PROGRAMOVÁNÍ V JAZYCE C
Typy dat
- byte (8 bitů), deklarace v asm instrukcí DB (Define Byte)
- word (16 bitů), deklarace DW (Define Word)
- dword (32 bitů), deklarace DD (Define Double-word)
- qword (64 bitů), deklarace DQ (Define Quad-word)
Jako operandy instrukcí akceptuje 80386 maximálně 32 bitů ve 32-bitových registrech (E??).
Některé instrukce pracují i se 64-bitovými slovy, ta se pak předávají v registrových párech EAX & EDX a EBX & ECX (vždy takto spolu).
.DATA
mstr db 'Hello', 0
xp db 100
icnt dw ?
ptrs dd 20 dup(0)
.CODE
...

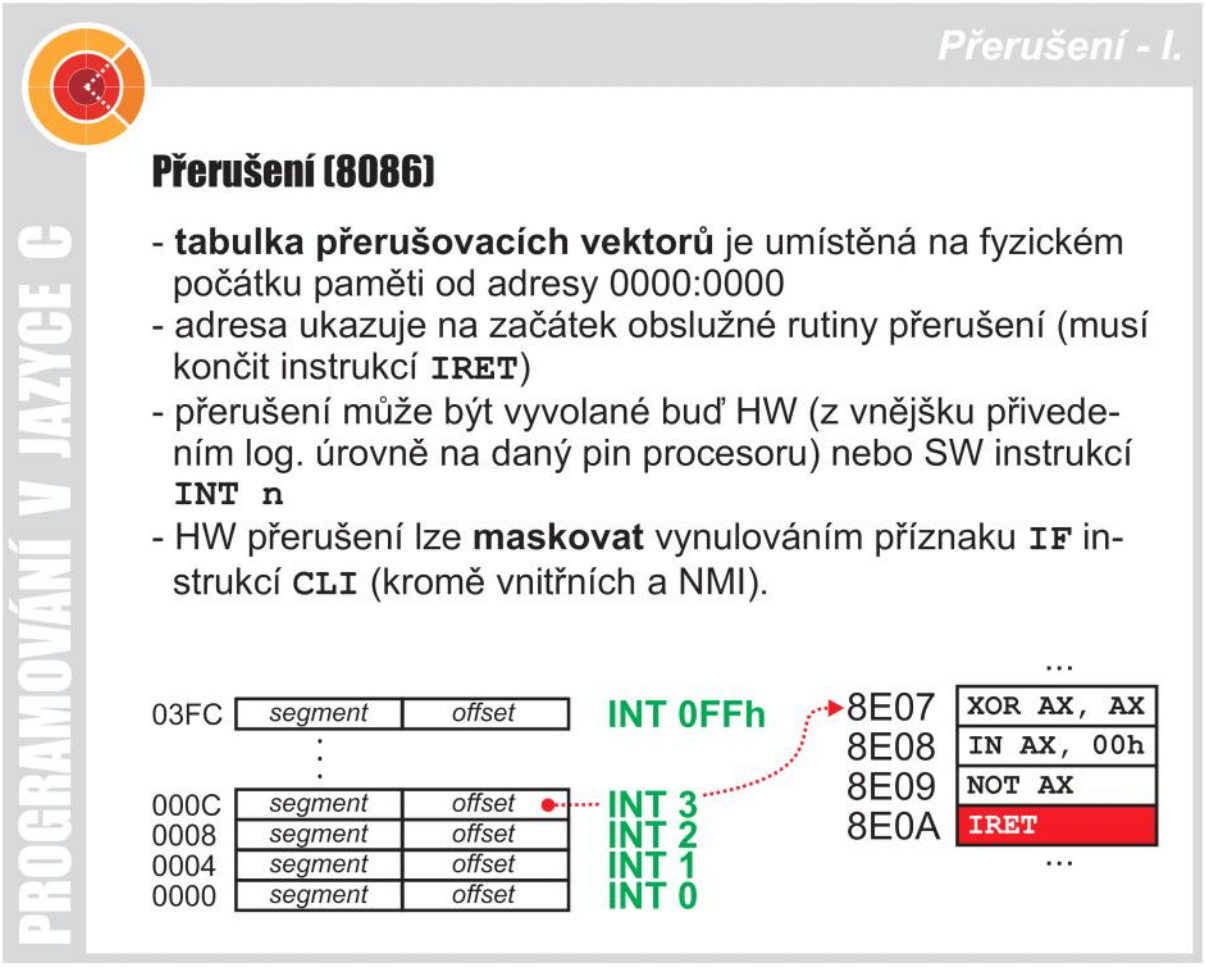
Endian procesoru (čili uspořádání bytů ve slovech)

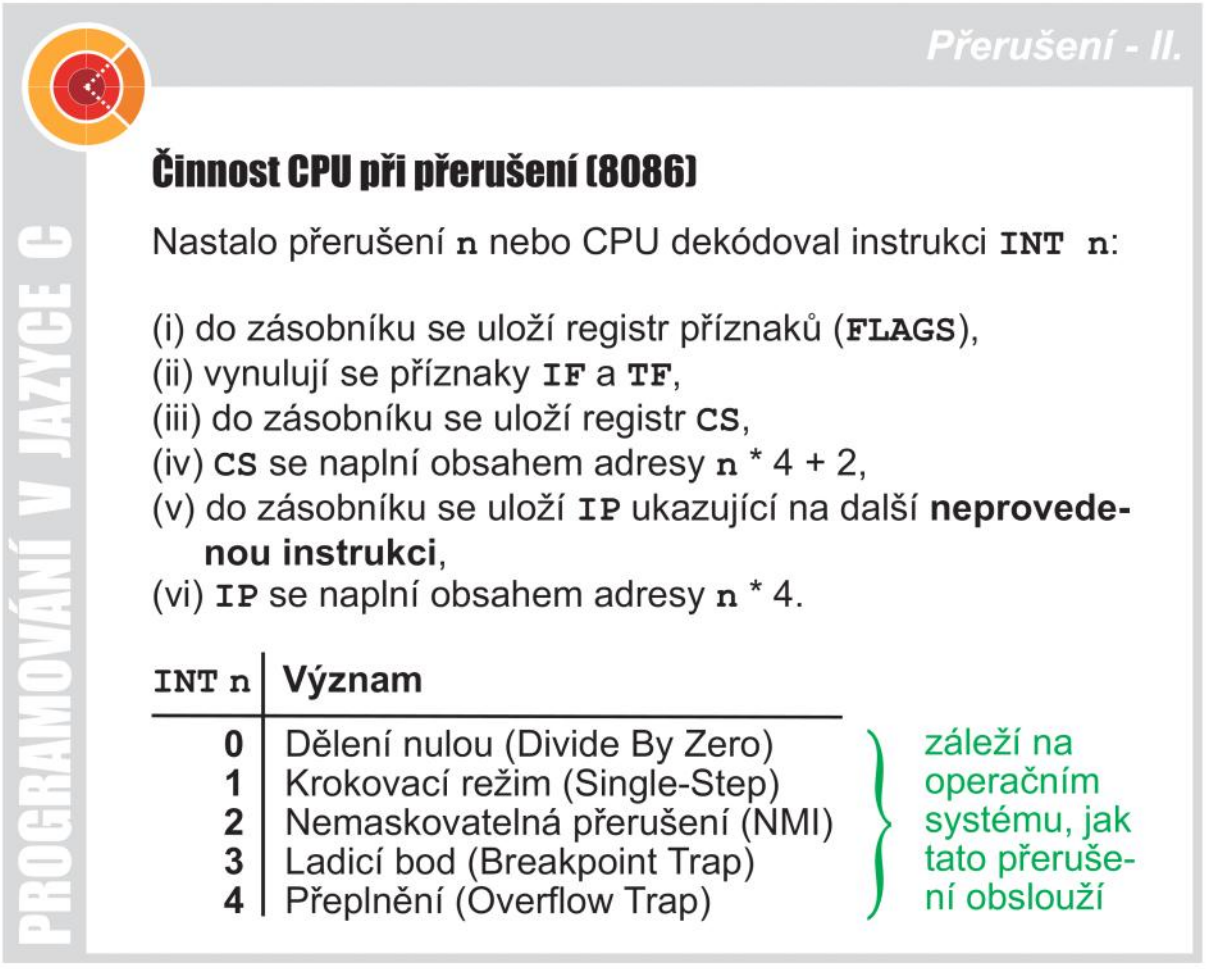
Procesory Intel (a jejich klony) používají **Little Endian**, tj. nižší řády jsou na nižších adresách:

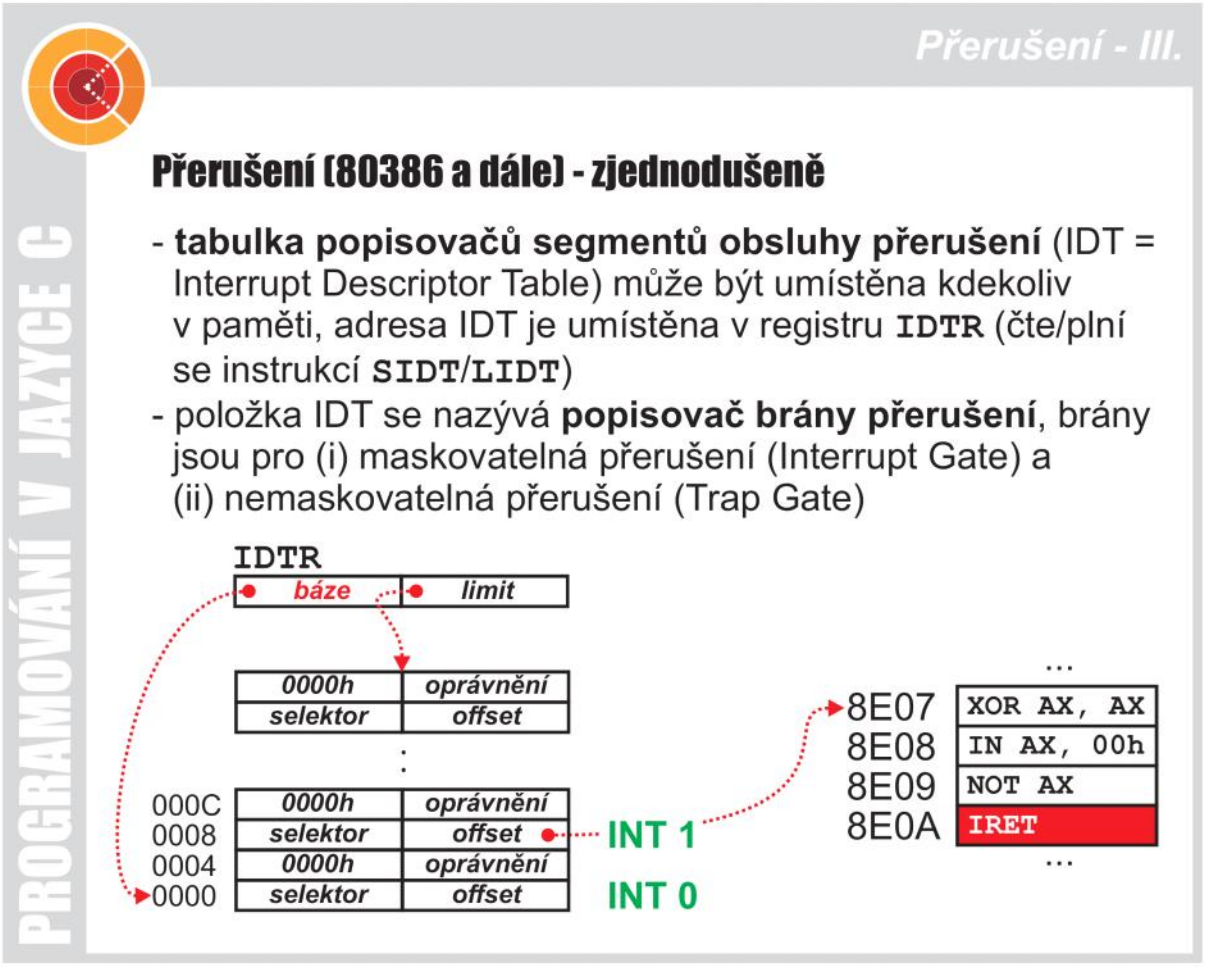


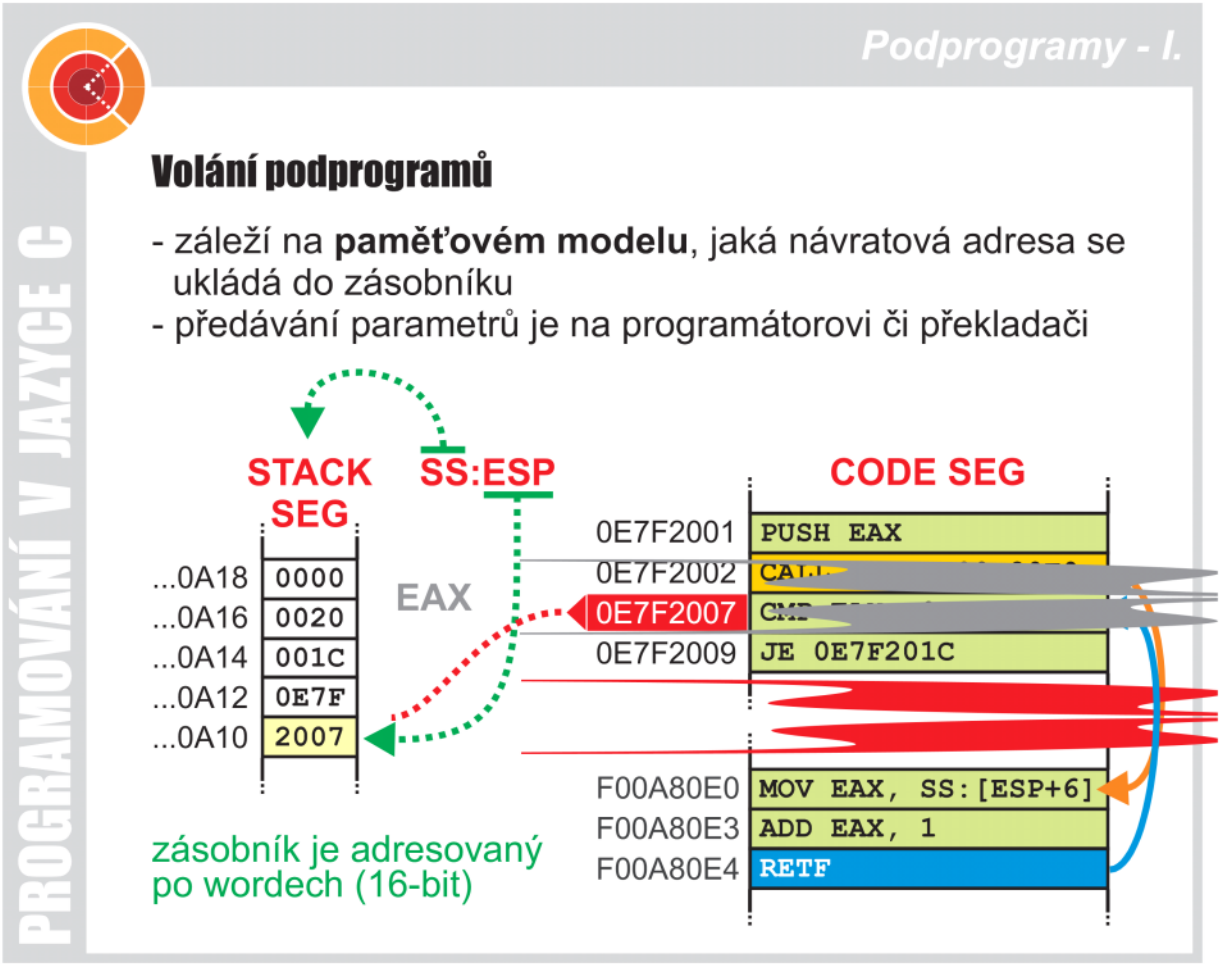
Procesory Motorola, AIM PowerPC (po G5), Sun SPARC (starší verze do V9), IBM System/370 používají **Big Endian**, tzn. nižší řády na vyšších adresách (vlastně tak, jak se číslo píše na papír).

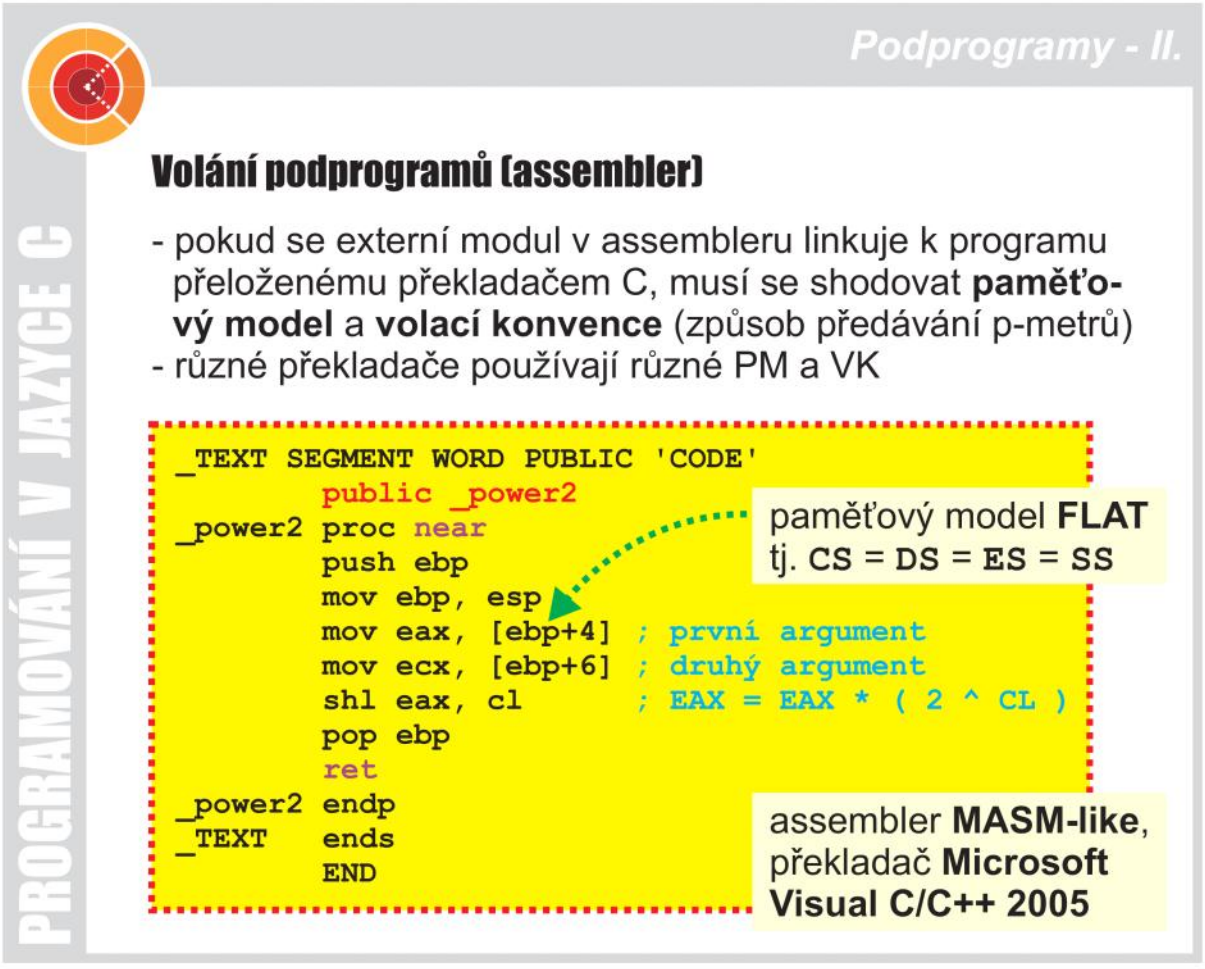
Některé procesory umí endian podle potřeby přepínat, např. ARM, SPARC V9, MIPS, PA-RISC, IA64, DEC Alpha, některé PowerPC => tzv. **Bi-Endian**.

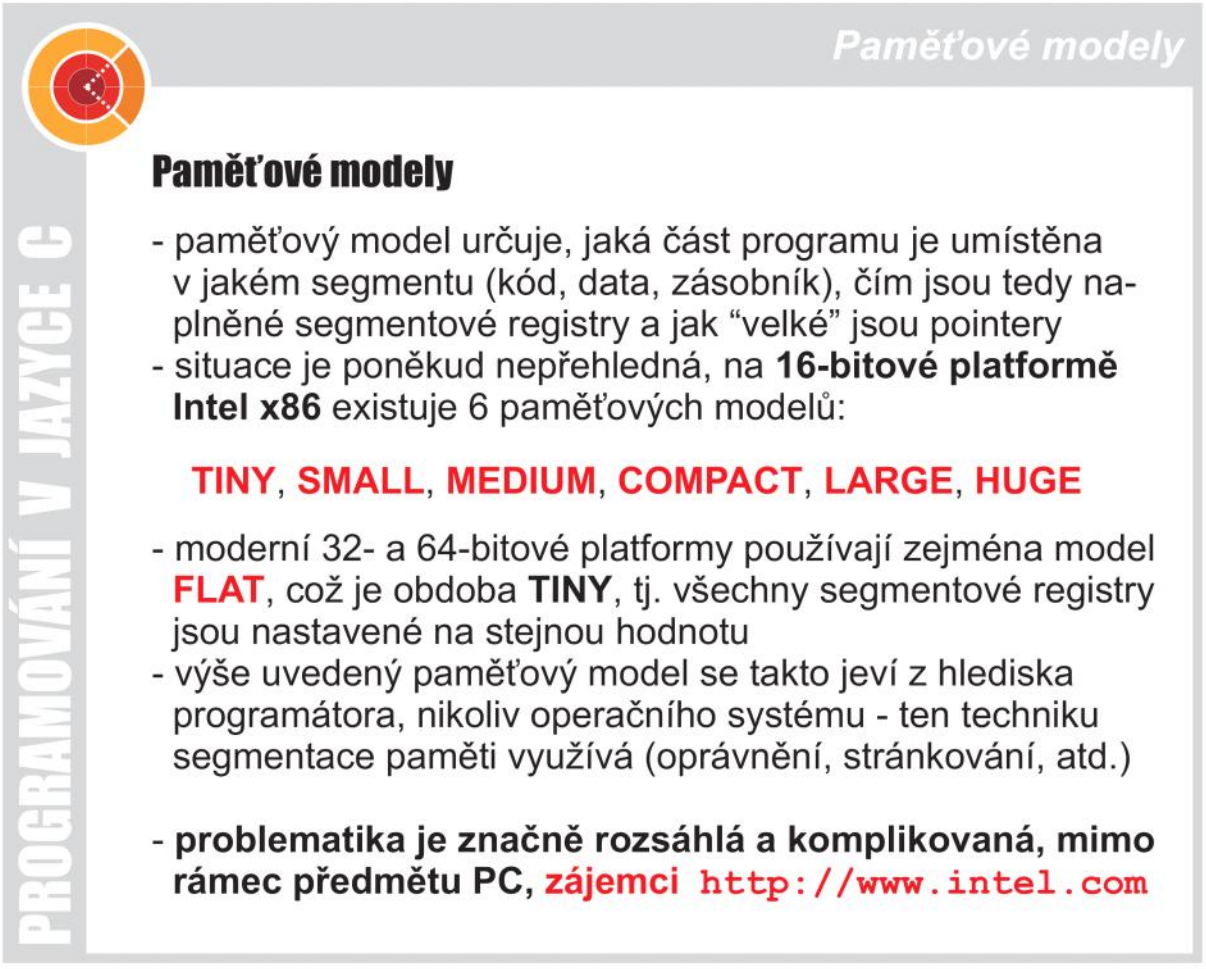

Přerušení - I.
PROGRAMOVÁNÍ V JAZYCE C
Přerušení (8086)
- tabulka přerušovacích vektorů je umístěná na fyzickém počátku paměti od adresy 0000:0000
- adresa ukazuje na začátek obslužné rutiny přerušení (musí končit instrukcí IRET)
- přerušení může být vyvolané buď HW (z vnějšku přivedením log. úrovně na daný pin procesoru) nebo SW instrukcí INT n
- HW přerušení lze maskovat vynulováním příznaku IF instrukcí CLI (kromě vnitřních a NMI).
03FC segment offset INT 0FFh
000C segment offset INT 3
0008 segment offset INT 2
0004 segment offset INT 1
0000 segment offset INT 0
8E07 XOR AX, AX
8E08 IN AX, 00h
8E09 NOT AX
8E0A IRET

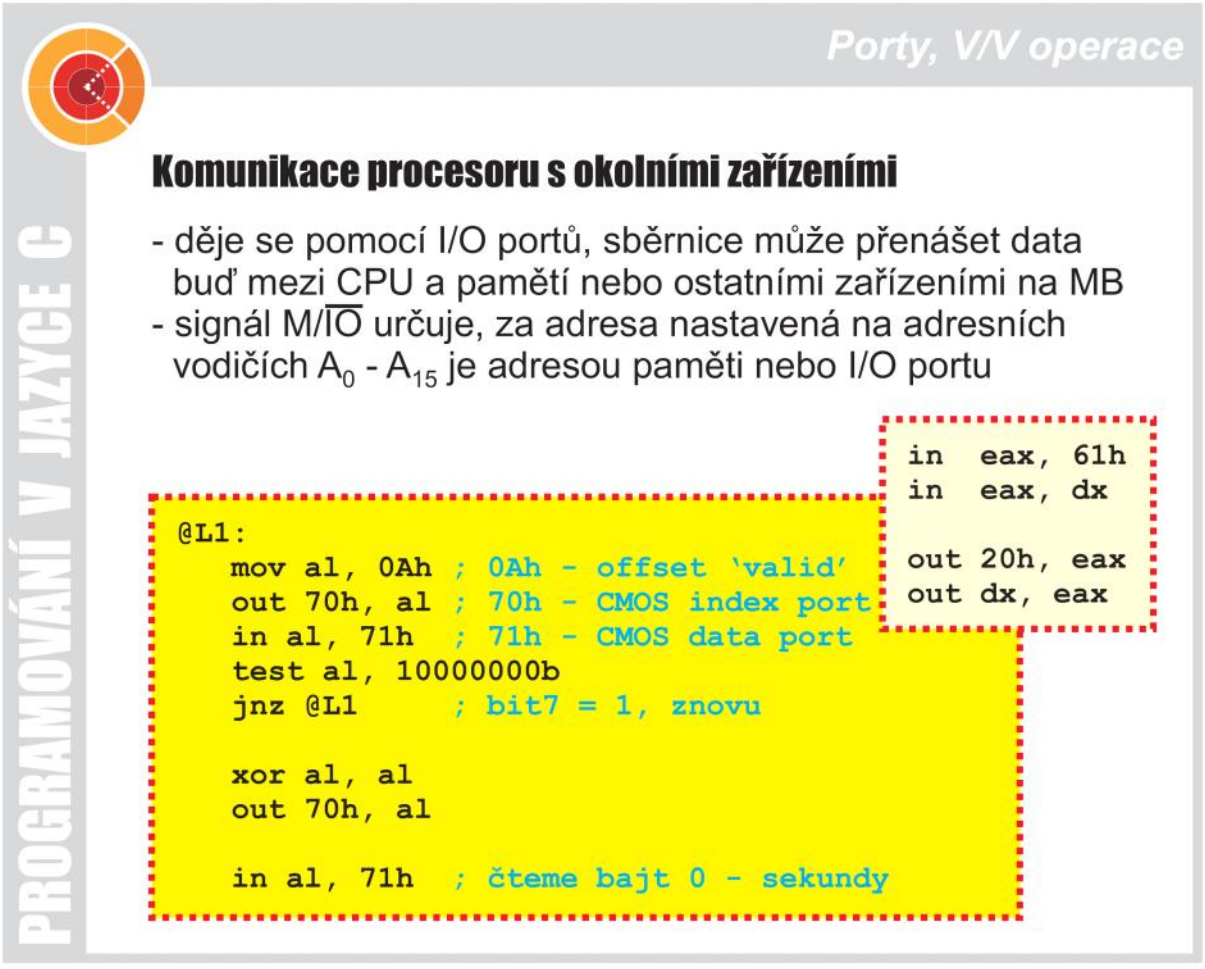

Přerušení - II.
PROGRAMOVÁNÍ V JAZYCE C
Činnost CPU při přerušení (8086)
Nastalo přerušení n nebo CPU dekódoval instrukci INT n:
(i) do zásobníku se uloží registr příznaků (FLAGS),
(ii) vynulují se příznaky IF a TF,
(iii) do zásobníku se uloží registr CS,
(iv) CS se naplní obsahem adresy n * 4 + 2,
(v) do zásobníku se uloží IP ukazující na další neprovedenou instrukci,
(vi) IP se naplní obsahem adresy n * 4.
INT n
Význam
0 Dělení nulou (Divide By Zero)
1 Krokovací režim (Single-Step)
2 Nemaskovatelná přerušení (NMI)
3 Ladicí bod (Breakpoint Trap)
4 Přeplnění (Overflow Trap)
záleží na operačním systému, jak tato přerušení obslouží


Přerušení - III.
PROGRAMOVÁNÍ V JAZYCE C
Přerušení (80386 a dále) - zjednodušeně
- tabulka popisovačů segmentů obsluhy přerušení (IDT = Interrupt Descriptor Table) může být umístěna kdekoliv v paměti, adresa IDT je umístěna v registru IDTR (čte/plní se instrukcí SIDT/LIDT)
- položka IDT se nazývá popisovač brány přerušení, brány jsou pro (i) maskovatelná přerušení (Interrupt Gate) a (ii) nemaskovatelná přerušení (Trap Gate)
IDTR
báze
limit
0000h
oprávnění
selektor
offset
000C
0008
0004
0000
INT 1
INT 0
8E07
8E08
8E09
8E0A
XOR AX, AX
IN AX, 00h
NOT AX
IRET


Podprogramy - I.
PROGRAMOVÁNÍ V JAZYCE C
Volání podprogramů
- záleží na paměťovém modelu, jaká návratová adresa se ukládá do zásobníku
- předávání parametrů je na programátorovi či překladači
STACK SEG
SS:ESP
CODE SEG
...0A18 0000
...0A16 0020
...0A14 001C
...0A12 0E7F
...0A10 2007
EAX
0E7F2001 PUSH EAX
0E7F2002 CALL
0E7F2007 CMP
0E7F2009 JE 0E7F201C
F00A80E0 MOV EAX, SS:[ESP+6]
F00A80E3 ADD EAX, 1
F00A80E4 RETF
zásobník je adresovaný po wordech (16-bit)


Podprogramy - II.
PROGRAMOVÁNÍ V JAZYCE C
Volání podprogramů (assembler)
- pokud se externí modul v assembleru linkuje k programu přeloženému překladačem C, musí se shodovat paměťový model a volací konvence (způsob předávání p-metrů)
- různé překladače používají různé PM a VK
_TEXT SEGMENT WORD PUBLIC 'CODE'
public _power2
_power2 proc near
push ebp
mov ebp, esp
mov eax, [ebp+4] ; první argument
mov ecx, [ebp+6] ; druhý argument
shl eax, cl ; EAX = EAX * (2 ^ CL)
pop ebp
ret
_power2 endp
_TEXT ends
END
paměťový model FLAT
tj. CS = DS = ES = SS
assembler MASM-like,
překladač Microsoft
Visual C/C++ 2005


Paměťové modely
PROGRAMOVÁNÍ V JAZYCE C
Paměťové modely
- paměťový model určuje, jaká část programu je umístěna v jakém segmentu (kód, data, zásobník), čím jsou tedy naplněné segmentové registry a jak "velké" jsou pointery
- situace je poněkud nepřehledná, na 16-bitové platformě Intel x86 existuje 6 paměťových modelů:
TINY, SMALL, MEDIUM, COMPACT, LARGE, HUGE
- moderní 32- a 64-bitové platformy používají zejména model FLAT, což je obdoba TINY, tj. všechny segmentové registry jsou nastavené na stejnou hodnotu
- výše uvedený paměťový model se takto jeví z hlediska programátora, nikoliv operačního systému - ten techniku segmentace paměti využívá (oprávnění, stránkování, atd.)
- problematika je značně rozsáhlá a komplikovaná, mimo rámec předmětu PC, zájemci http://www.intel.com


Porty, V/V operace
PROGRAMOVÁNÍ V JAZYCE C
Komunikace procesoru s okolními zařízeními
- děje se pomocí I/O portů, sběrnice může přenášet data buď mezi CPU a pamětí nebo ostatními zařízeními na MB
- signál M/IO určuje, za adresa nastavená na adresních vodičích A0 - A15 je adresou paměti nebo I/O portu
in eax, 61h
in eax, dx
out 20h, eax
out dx, eax
@L1:
mov al, 0Ah ; 0Ah - offset 'valid'
out 70h, al ; 70h - CMOS index port
in al, 71h ; 71h - CMOS data port
test al, 10000000b
jnz @L1 ; bit7 = 1, znovu
xor al, al
out 70h, al
in al, 71h ; čteme bajt 0 - sekundy

Implementace režimu jádra a uživatelského režimu.

Thursday, May 30, 2013 8:27 AM

While transitioning from user mode to kernel mode, the processor makes a switch between the per-process-user-stack and the per-process-kernel-stack. Then the user-per-process stack segment selector and stack pointer is stored in the kernel stack and then the `eip` instruction pointer (return address at user mode) and other hardware registers are pushed on to the kernel stack
When the kernel has to return to user mode, the `trapret` code pops all values stored in the kernel stack back to the hardware registers.

From <http://stackoverflow.com/questions/9968028/returning-from-kernel-mode-to-user-mode>

Výpočet v módu jádro – v důsledku událostí:

Popsat rozdíl mezi uživatelským módem a režimem jádra.

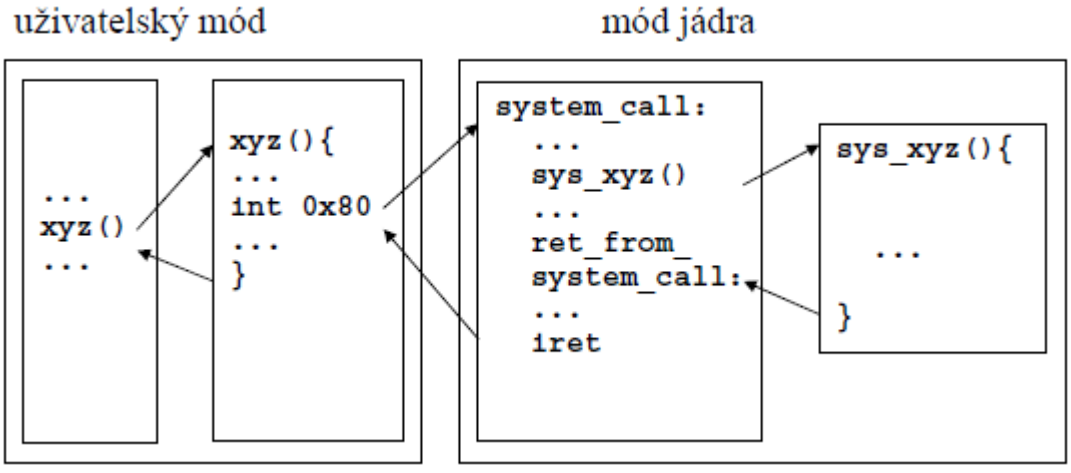
- Přerušení
- Výjimky
- Softwarové přerušení

Řízení se předá na proceduru pro ošetření odpovídající události. Část stavu přerušeného procesu potřebná pro jeho vykonávání po skončení obsloužení události (počítadlo instrukcí, PSW (**Processor Status Word**) se uloží do zásobníku jádra přerušeného procesu.

Systémové volání

V standardní knihovně jazyka C je pro každé systémové volání obálková procedura, řízení se předá software přerušením proceduře jádra, která se nazývá syscall(), system_call. Protože je jen jedna pro všechny služby, požadovaná služba je identifikována parametrem procedury, který se nazývá číslo systémového volání.

Linux



Aplikační program volá službu xyz, obálková procedura uloží číslo služby do registru **eax** před vykonáním **int 0x80**.

System_call

- Uloží obsah registrů (hardwarový kontext)
- Zavolá odpovídající funkci (v jazyce C)
- Ukončí se voláním ret_from_sys_call()

Výjimky se zpracují obdobně

- jsou synchronní s procesem (vznikají v důsledku událostí způsobených vykonáváním procesu)

- Procedury pro jejich zpracování mají obdobnou strukturu jako procedura pro systémová volání **systém_call**

Linux

Procedura pro zpracování výjimky

- Uloží obsah registrů
- Zpracuje výjimka (funkce v jazyku C)
  - Pošle signál procesu
  - Zpracuje žádost o stránku
- Ukončí se voláním funkce ret_from_exception()

Zpracování přerušení

Přerušení je obecně asynchronní vzhledem k přerušenému procesu – proces čeká na přenos dat, po dokončení přenosu je přerušen úplně jiný proces. Zpracování přerušení nesmí způsobit čekání, přerušený proces zůstává ve stavu běžící. Čas zpracování přerušení je započítán přerušenému procesu, při zpracování přerušení se tedy přistupuje do jeho záznamu **proc**.

Obsloužení přerušení:

- Uloží IRQ (Interrupt ReQuest) a obsah registrů
- Pošle potvrzení PIC (Programmable Interrupt Controller)
- Vykoná obslužní proceduru přerušení
- Ukončí se skokem na ret_from_intr()

Vzájemné vnoření systémového volání, výjimek a přerušení

Předpokládejme odladěné jádro

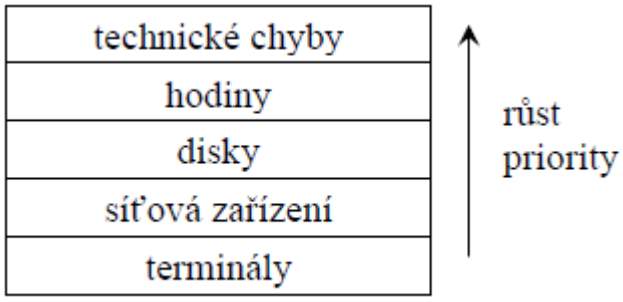
1. Zpracování systémového volání
   - Může vzniknout výjimka při žádosti o stránku (výpadek stránky)
   - Může dojít k přerušení
2. Zpracování výjimky
   - Může dojít k přerušení
3. Zpracování přerušení
   - Může dojít k přerušení

Při každém odkladu zpracování některé z uvedených událostí musíme nejdříve uložit odpovídající HW kontext.

- Vytváří se kontextové vrstvy zásobníku jádra přerušeného procesu
- Existuje globální zásobník přerušení

Prioritní schéma

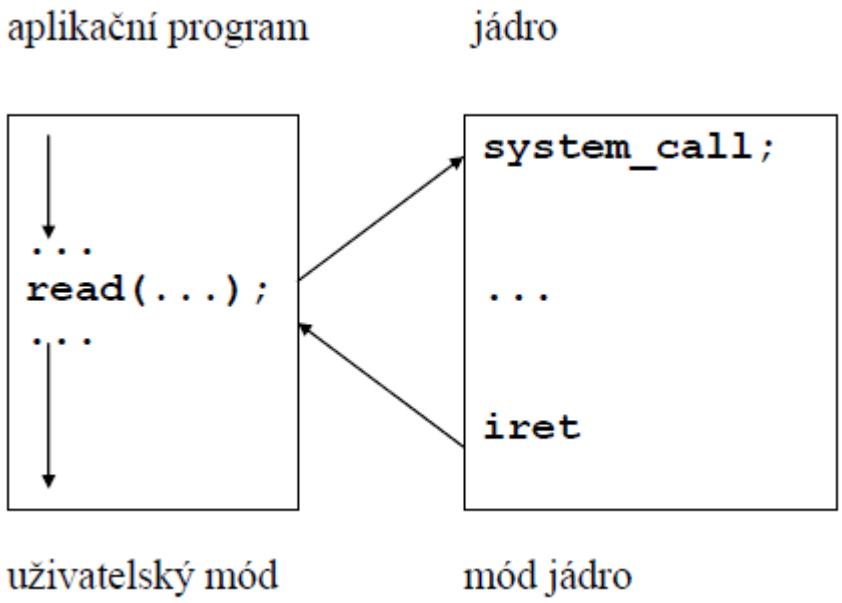
- Přerušení mají přiřazené prioritní úrovně (interrupt priority level)



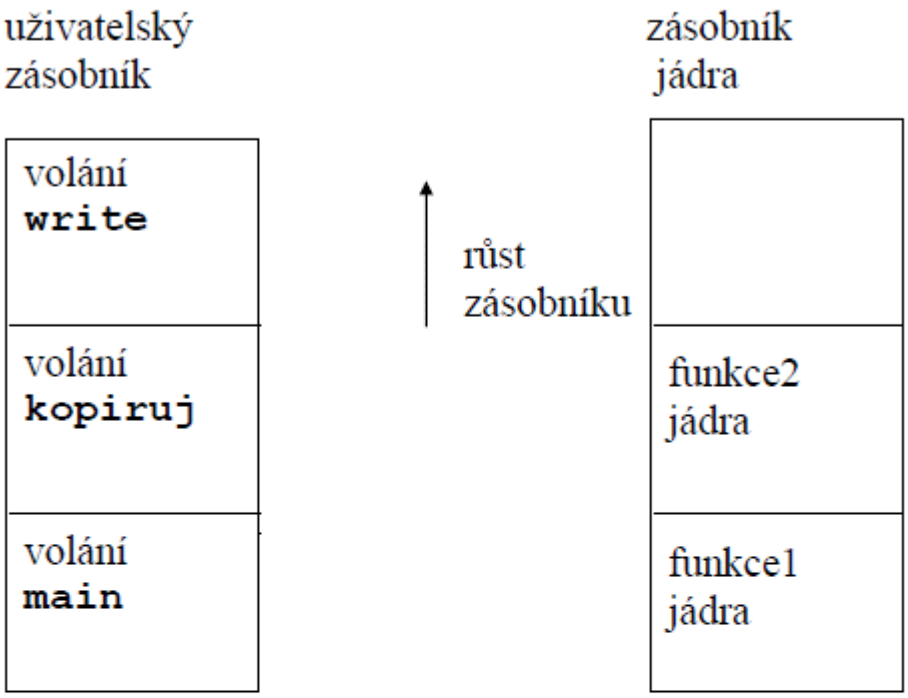
Ve stavovém registru procesoru je nastavena okamžitá prioritní úroveň zpracovávaného přerušení.
Vznikne-li přerušení přerušení s nižší nebo stejnou prioritní úrovní, je uloženo a jeho obsluha je odložena.
Vznikne-li přerušení s vyšší prioritní úrovní, uloží se HW kontext, na tuto vyšší prioritní úroveň se nastaví hodnota okamžitého přerušení ve stavovém registru procesoru a přerušení se zpracuje.

Při skončení zpracování přerušení se z uloženého PSW obnoví okamžitá prioritní úroveň přerušení.

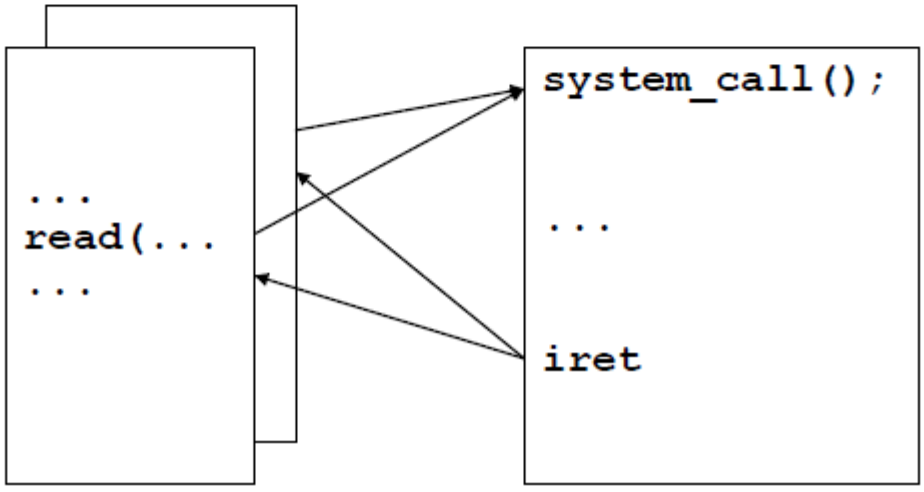
Proces a jádro



- uživatelský
- jádra



více procesů



Jádro je reentrantní – každý proces má svůj zásobník jádra, často v adresovém prostoru procesu – chráněný, spravovaný jádrem. Každý proces má položku v tabulce procesů: **proc** záznam a **u** (user) oblast.

Z PPR

Princip – MSDOS compatible, uniprocesor

V reálném režimu: Pomocí služby DOSu se nainstaluje handler přerušení 8, které generují hodiny. Když je volána obsluha přerušení, v zásobníku jsou uloženy registry CS, IP a Flags kódu, jehož vykonávání bylo přerušeno. Obsluha přerušení uloží stávající registry, plánovač vybere novou úlohu a zapíše do zásobníku její hodnoty uvedených registrů (CS, IP a Flags). Obsluha přerušení obnoví zbývající registry procesoru pro plánovačem vybranou úlohu. Provede se instrukce iret, kterou se spustí naplánovaný proces díky přepsání hodnot v zásobníku.

Proces a thread - stavy, implementace, plánování, synchronizace.

Thursday, May 30, 2013 8:27 AM

Wiki: Thread je nejlehčí jednotka plánování, TXKoutný: Proces - *největší* výpočetní entita plánovače.

Je potřeba rozlišovat thread programátorský a thread z hlediska plánovače – dvě úplně odlišné věci, ale v literatuře se oboje jmenuje thread a pak jsou z toho zmatky.

- Viz přednáška 6 OS

Typy vláken dle OS
- jádrová
- lehké procesy
- uživatelská

Process

- Má svůj vlastní kontext a adresní prostor
- Může mít jedno nebo více vláken v jednom (svém) adresním prostoru
- Vlákna uvnitř procesu sdílejí prostředky procesu (otevřené soubory)

Thread

- Kernel thread: jednotka plánování plánovače jádra (činnost plánovače je založená na přepínání kernel threadů), jejich počet je nezávislý na počtu jader procesoru nebo počtu procesorů
  - pouze jádrová věc
- User thread: uživatelem vytvořený thread v rámci prostředků programovacího jazyka
  - Bez podpory plánovače jádra: fiber (100% user space)
  - S podporou plánovače jádra: lightweight proces (lehký protože nemá vlastní adr. prostor, ukazatele na soubory apod.) – fiber namapovaný na kernel thread
- Sdílená paměť a Thread-local storage (může mít a nemusí, v rámci adresního prostoru procesu)

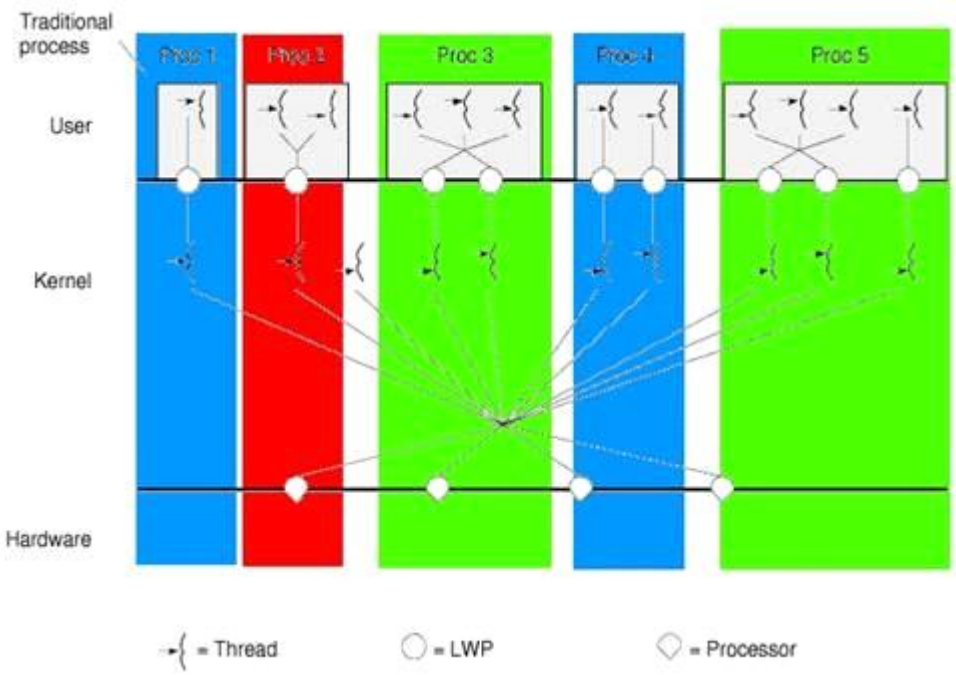
Fiber

- Běží v uživatelském prostoru a při přepnutí fiberů se nepřepíná kontext – rychlé a levné
- Spolupracují kooperativně (ne preemptivně) – musí udělat yield, jsou **implicitně synchronizované**
- Méně problematická thread-safety: nejsou potřeba spinlocky a atomické operace
- Vyžaduje menší podporu od OS, nemusí požadovat vůbec žádnou (třeba podle výhodnosti plánování –OS může a nemusí mít „lepší“ plánovač). Podporují je Unixové systémy, Microsoft, Symbian…
- Nemohou využívat víc procesorů (všechny fibery jsou v jediném kernel threadu)
- Sdílená paměť a Fiber-local storage

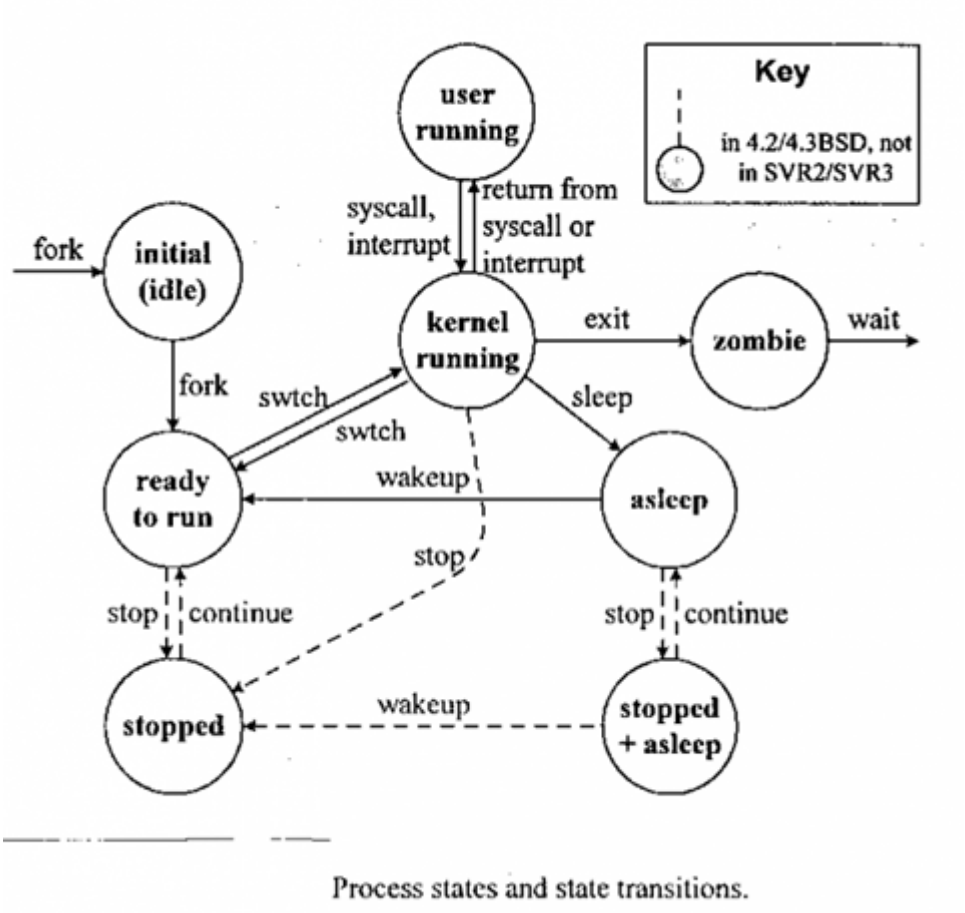
LWP (Light-Weight Process)

- Linux: http://www.linuxforu.com/2011/08/light-weight-processes-dissecting-linux-threads/
  - Kernel thread = LWP
  - Obsluha user-thread (u Linux = process) na LWP 1:1
- Na Windows: http://www.i.u-tokyo.ac.jp/edu/training/ss/lecture/new-documents/Lectures/03-ThreadScheduling/ThreadScheduling.pdf
  - Windows plánuje vlákna, ne procesy
  - Plánování je preemptivní, založeno na prioritě a používá round-robin pro nejvyšší priority
  - Každý thread má aktuální a základní prioritu - základní priorita je inicializována při spuštění procesu, aktuální pak závisí na chování threadu - priorita se snižuje, pokud thread vždy vyčerpá přidělené kvantum

Vláknové modely

- **1:1 (kernel vlákna)** Vlákna vytvořená uživatelem odpovídají počtem 1:1 entitám, které plánuje jádro. Nejjednodušší přístup, ale je třeba uvážit preempci a thread-safety
- **N:1 (uživatelská vlákna)** všechna aplikační vlákna jsou namapována na jedno jádrové vlákno, jádro nemá tušení o tom, že aplikace běží vláknově.
- **M:N (hybridní)** komplexní na implementaci (vyžaduje změny v jádře i uživ. prostoru) ale umožňuje zvýšení efektivity výpočtu (například přiřazení více uživatelských vláken více vláknům jádra – proces může běžet na více procesorech).
- 

Stavy procesu



Process states and state transitions.

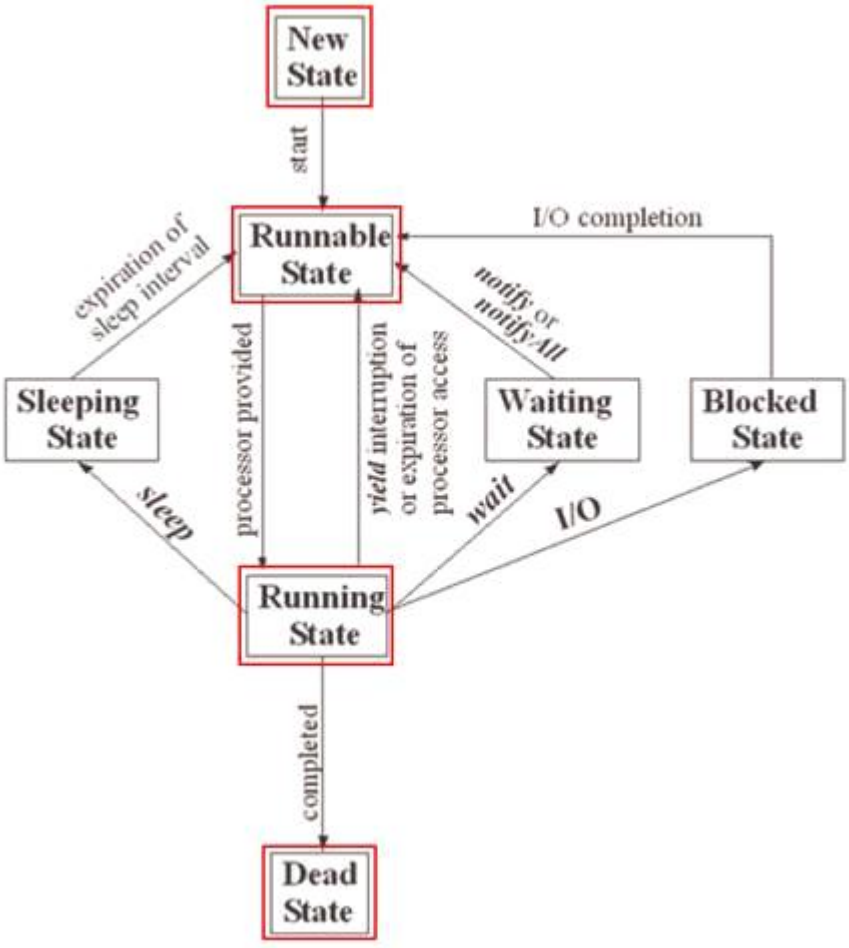
- **Počáteční** (initial/idle) – fork začal vytváření procesu.
- **Připraven na vykonání** (ready to run) – fork dokončil vytváření procesu, proces čeká až bude

naplánován na přidělení procesoru.

- **Běžící v jádře** (kernel running) – byl naplánován, vykoná se přepnutí kontextu, procedura jádra swtch() uloží HW kontext do registrů.
- **Běžící uživatelsky** (user running) – proces vykonává svůj programový kód. Může přejít do stavu běžící v jádře v důsledku volání služby jádra nebo přerušení, po skončení obsluhy se vrátí.
- **Spící** (asleep) – při vykonávání systémového volání se může stát, že je nutno čekat na nějakou událost nebo prostředek, proces (v jádře) proto zavolá proceduru sleep(). Když událost nastane, jádro vzbudí proces, proces se stane připraven na vykonání a po naplánování pokračuje obsluha systémového volání ve stavu běžící v jádře.

  Připraven na vykonání se může stát také, je-li běžící a uplyne mu přidělené časové kvantum - vykoná se preempce běžícího procesu a to ve stavu běžící uživatelsky nebo při návratu do něj. Jádro je nepreemptivní. Přerušení se může vyskytnout i ve stavu běžící v jádře, kdy po skončení obsluhy přerušení proces pokračuje ve stavu běžící v jádře.

- **Mátoha** (zombie) - proces končí voláním exit() anebo v důsledku signálu, dokud rodič nevykoná wait().
- **Zastaven** (stopped) - do něj přejde proces, je-li běžící nebo spící (+ asleep), po stop signálech:
  - SIGSTOP zastav proces
  - SIGTSTP CTRL-Z
  - SIGTTIN tty čtení procesu v pozadí
  - SIGTTOU tty psaní procesu v pozadí
  - SIGCONT převede proces do stavu připraven na vykonání nebo do stavu spící.



Plánování, plánovací třídy, inverze priority

Víceúlohové systémy:

- systémy reálného času
- interaktivní systémy

- dávkové systémy

  Ve víceúlohových systémech sdílení času je tradiční problém „vhodně“ a spravedlivě přidělovat čas jednotlivým procesům (adekvátně k typu – rt/dávkový) a zajišťovat vysokou průchodnost (tyhle tři požadavky jsou často v kontradikci). Je třeba hlídat aby nedošlo k vyhladovění a k inverzi priorit.

  Problém velikosti časového kvanta: v interaktivních systémech je třeba přepínat často, aby byl vytvořen dojem okamžité odezvy, ale přepínání HW kontextu je náročné a zdržuje = čím kratší čas, tím větší režie systému.

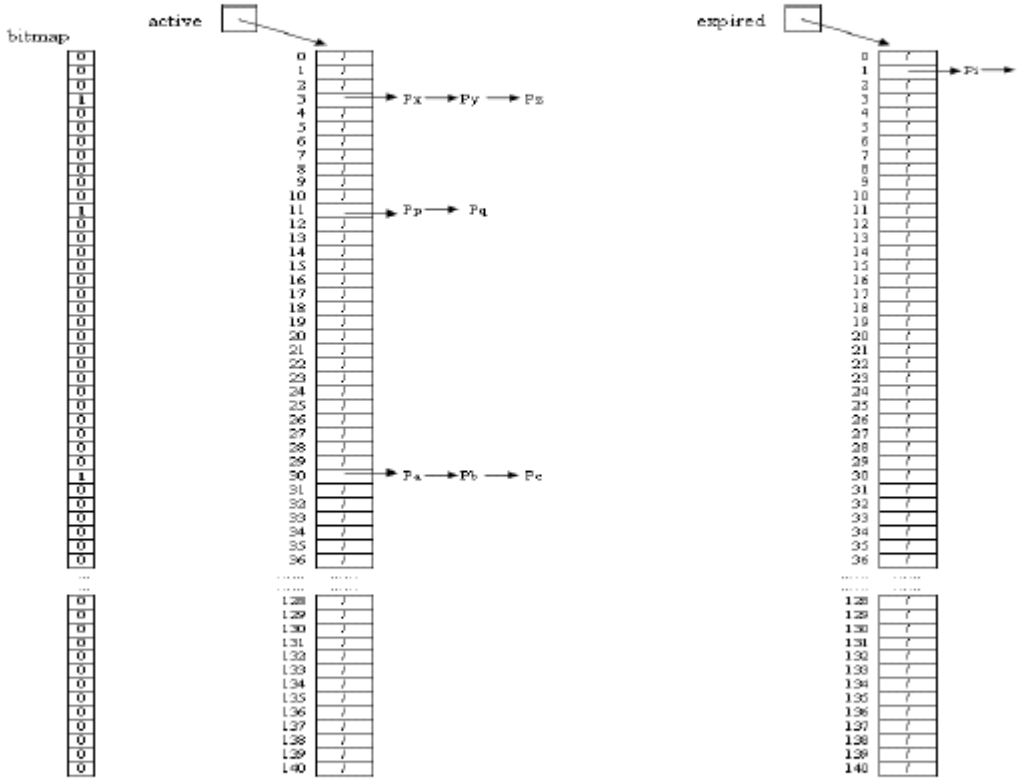
  Typy plánování:

- preemptivní – s předbíháním, proces je možno přerušit zvenčí, je potřeba zavést synchronizační primitiva
- nepreemptivní – proces se musí vzdát procesoru sám (fiber)

  Plánování:

- **Linux 2.6 – plánovač *O(1)* / Ingo Molnar**

  - fronta připravených procesů pro každý procesor
  - fronta je složena ze dvo polí o velikosti počtu priorit
    active – procesy s ještě nevyčerpaným časovým kvanten
    prioritní plánování, RR v rámci priority
    expired- procesy s vyčerpaným časovým kvantem

  nové časové kvantum a priorita se vypočte ihned po vyčerpání předcházejícího kvanta *O(1)* a proces se zařadí do fronty procesů s novou prioritou do pole **expired** na index odpovídající priorité
  je-li pole **active** prázdné „konec epochy“ pole se přepnou

- **FIFO** – nejjednodušší, neadaptabilní, málokdy spravedlivý. Když nějaký proces nedoběhne, může

dojít k vyhladovění

- **Shortest proces (job) first -** *nepreemptivní*, u dávkových úloh, předpokládá se znalost dob trvání procesů, vezme se proces/job s nejkratší předpokládanou dobou běhu, po jeho skončení se z fronty bere další proces; optimalizuje dobu obrátky - dávkové
- **shortest remaining time –** *preemptivní,* vybere úlohu, jejíž zbývající doba běhu je nejkratší - když je plánovaná úloha s 10 min do jejího dokončení a přijde úloha trvající 1 minutu, systém provede přeplánování a začne běžet ta nová úloha; dobrý pro krátký (hl. I/O vázaný) úlohy, může ale dojít k vyhladovění (v systému je hodně „krátkých" úloh, na ty dlouhodobější se nedostane řada)
- **Round robin** – procesy obdrží každý stejný čas a střídají se dokola, nedojde k vyhladovění protože není zavedena priorita
- **Prioritní plánování** – Každý proces má svou prioritu (procesy jádra nejvyšší, interaktivní vysokou, dávkové nízkou), procesy jsou podle priority roztříděny do tříd, ve kterých se periodicky střídají metodou Round Robin (vždy nejdřív první neprázdná třída od nejvyšší priority). Může dojít k vyhladovění procesů s nízkou prioritou (řešením je zvyšovat prioritu dlouho nenaplánovaným procesům).
    - priorita **statická** (při startu procesu) a **dynamická** (chování procesu v poslední době, snižuje ji u běžícího procesu při každém tiku plánovače; = 1/f, *f* je velikost částí kvanta, kterou proces naposledy použil)
- Prakticky ve všech dnešních OS, mnoho různých metod:
- **Fair Share** - Férové rozdělení času mezi uživatele, nikoli procesy. Rekurzivní aplikace Round Robin na každé úrovni abstrakce - nejprve na skupiny, pak na uživatele, pak na procesy.
- **Loterie** - plánovač má k dispozici pevný počet tiketů, každý proces obdrží určitý počet tiketů a plánovač pak vybere jeden tiket - **náhodně** vybere tiket a přidělí časové kvantum procesu, který ten tiket vlastní
- **Epochy** - čas procesoru je rozdělen do epoch
    a. každý proces má specifikováno časové kvantum v rámci epochy
    b. v jedné epoše proces může využívat své časové kvantum po částech
    c. epocha končí, když všechny běhu schopné procesy vyčerpaly svá časová kvanta
    d. Délka časového kvanta v epoše závisí na prioritě procesu.

Jaký je rozdíl mezi NoRealTime, SoftRealTime a HardRealTime

NoRealTime – Nemají žádný deadline.

SoftRealTime – překročení termínu se toleruje, systém reaguje zhoršenou kvalitou poskytovaných služeb – např. vypadne pár snímků, nebo přílet letadla se dozvíte s několikasekundovým zpožděním.

HardRealTime – Dokončení výpočtu po termínu se považuje za chybu a výsledek za bezcenný – strict deadline. Nedodržení termínu může vést k celkovému selhání systému (airbag, jaderná elektrárna…).

Z <https://d.docs.live.net/67cbdb2faf0647ea/Dokumenty/FAV/A11N0110P/Státní%20závěrečná%20zkouška/PPR/PPR_Karfik_v2.docx>

- Soft RT
- Hard RT

Viz níž.

Inverze priority

Inverzí priorit rozumíme situaci, která nastane, pokud vlákno s vyšší prioritou požaduje přístup k systémovým zdrojům, které v danou chvíli právě exkluzivně drží vlákno s nižší prioritou. V tomto případě dojde k preempci do vlákna s vyšší prioritou, které však nemůže běžet díky zablokovanému systémovému zdroji. To je velice nepříjemná situace, zejména v RTOS. Jediným řešením je umožnit vláknu, které systémový zdroj drží co nejrychleji doběhnout a umožnit tak i jiným vláknům pokračovat v jejich činnosti. **K vyřešení této situace se používá systém inverze priorit**, který umožní vláknu s nižší prioritou zdědit prioritu kritického vlákna, rychle vykonat potřebné operace až do chvíle uvolnění požadovaného systémového zdroje a dále pak nechat pokračovat v práci kritické vlákno.

Synchronizace

Kernel space (viz kapitola 9), vs. user space. a další (?)

Process Control Block (PCB)

Udržuje informace spojené s každým procesem, je využit při znovu rozběhnutí přerušeného procesu.

- Stav procesu
- Program counter
- CPU registry
- Informace o CPU plánování
- Info o správě paměti
- Info o status I/O (File descriptory, sockety atd.)

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Signály.

Thursday, May 30, 2013 8:28 AM

- Jen u Linuxu, Windows mají "alternativu" - message
- Signály jsou přerušení generována softwarem, která jsou zaslána procesu, pokud nastane nějaká událost

  Unix systémy používají signály, aby daly **vědět procesu, že nastala určitá událost** (proces je pak např. vzbuzen, přerušen, …). Oznamují se jen čísla signálu, žádné parametry.

Signál vs. přerušení

Hezky podané na:

- http://stackoverflow.com/questions/13341870/signals-and-interrupts-a-comparison
- Na *přerušení* jde pohlížet jako na prostředek komunikace *mezi procesorem a jádrem OS*
- *Signály* mohou být brány jako prostředek komunikace *mezi jádrem OS a procesy*
  - Mohou být započaty jádrem OS (SIGSEGV, SIGIO) nebo procesem (*kill()*)
  - Jsou nakonec spravovány jádrem OS, které je doručí do cílového procesu/vlákna a spustí buď nějakou obecnou akci (ignorovat, ukončit, ukončit + dump core) a nebo obsluhu signálu, kterou poskytl proces

Synchronní a asynchronní signály

Signály mohou být buď **synchronní nebo asynchronní**, záleží na zdroji a důvodu, proč byla událost signalizována.

Mezi synchronní signály patří např. neoprávněný přístup do paměti a dělení nulou. Pokud běžící program provede některou z těchto akcí, vygeneruje se signál. Synchronní signály jsou doručeny stejnému procesu, který provedl operaci, která ten signál způsobila (což je důvod, proč jsou považovány za synchronní).

Když je signál generován událostí, která je externí vzhledem k běžícímu procesu, tak ten proces přijme ten signál asynchronně. Příklady takových signálů jsou ukončení procesu pomocí konkrétní klávesové kombinace (třeba <control><C>) a vypršení časovače. Asynchronní signál je typicky zaslaný jinému procesu.

Obsluha signálu

Jakmile byl signál generován výskytem nějaké události (např. dělením nulou, neoprávněným přístupem do paměti, uživatel stisknul CTRL+C = SIGINT signál), signál je dopraven procesu, kde musí být zpracován. Proces, který přijme signál, jej může zpracovat různými způsoby:

- Ignorování signálu (kromě SIGKILL a SIGSTOP signálů)
- Použití defaultního (výchozího) signal handleru (obsluhy signálu/funkce na zpracování signálu dle Košičana)
- Poskytnutí vlastní signal-handling funkce = vlastní obsluha signálu (kromě SIGKILL a SIGSTOP).

Každý signál má svou **výchozí obsluhu signálu (*default signal handler*),** která běží v jádře, když je signál zpracováván **(???**that is run by the kernel when handling that signal). Tato výchozí akce může být přepsána **uživatelem definovanou obsluhou signálu**, který je volán pro obsluhu daného signálu. Signály mohou být obslouženy různými způsoby. Některé signály (jako změna velikosti okna) mohou být jednoduše ignorovány; jiné (jako třeba neoprávněný přístup do paměti) mohou být obslouženy ukončením programu.

Signály mohou být obslouženy nastavením určitých proměnných v C struktuře `struct sigaction` a pak předáním této struktury `sigaction()` funkci. Signály jsou definovány v include souboru `/usr/include/sys/signal.h`. Např. signál SIGINT reprezentuje signál pro ukončení programu pomocí <Control> <C>. Výchozí obsluha signálu (signal handler) pro SIGINT je

ukončit program.

Další možností je, že v programu může být nastavena vlastní funkce pro obsluhu signálu, a to nastavením `sa_handler` proměnné ve struktuře `struct sigaction` na název funkce, která obslouží ten signál, a pak zavoláním funkce `sigaction()`. Té se předají jako parametry (1) signál, pro který nastavujeme obsluhu, a (2) pointer na `struct sigaction`.

Fáze signálů

Všechny signály, ať už synchronní nebo asynchronní, mají stejný životní cyklus:

1. Signál je vygenerován a **odeslán** poté, co nastane nějaká událost.
   - PM (*Process manager*) nejprve zjistí, které procesy mají obdržet signál
     - V tabulce procesů je pro každý proces několik `sigset_t` proměnných (=bitmapy, definují ignorované, zachycované signály)
   - Pro procesy zachycující daný signál:
     1) jádro zaznamená v záznamu proc (deskriptoru procesu) cílového procesu odeslání nového signálu.
     2) Jádro přeruší standardní provádění posloupnosti instrukcí cílového procesu, uloží informace o stavu procesu, aby se pak mohl opět pokračovat v běhu. Informace jsou uloženy na zásobníku toho procesu (kterému má dorazit signál) + kontrola, že je dost místa na zásobníku (dělá PM).
   - PM pak volá system task in jádře pro uložení informací do zásobníku. System task také manipuluje s program counterem procesu, aby proces mohl být spuště v kódu obsluhy.
2. Vygenerovaný signál je **doručen – přijat -** procesu.
3. Jakmile je signál doručen, musí být zpracován.
   - Když obsluha skončí, je provedeno systémové volání `sigreturn`. Prostřednictvím tohoto volání se PM i jádro podílejí na obnově kontextu signálu a registrů „signalizovaného“ procesu, aby mohl pokračovat v normálním běhu.
   - Pokud není signál zachycen (nemá def. handler), podnikne se defaultní akce, která se může týkat volání souborového systému pro vytvoření **core dumpu** (zápis obrazu paměti procesu do souboru, který může být prozkoumán debuggerem) či zabití procesu, pro něž je třeba zapojit PM, souborový systém a jádro.
   - PM řídí jednu nebo více opakování akcí výše - podle toho, jestli je signál doručen jednomu procesu nebo skupině procesů.

Fáze vypořádání se signály jinak:

1. Příprava (Preparation) – kód programu se připraví pro možný signál
   - Několik systémových volání, která lze nastavit jako odpověď na signál
   - `sigaction` → co má proces dělat se signálem: ignorovat/zachytit/nastavit defaultní reakci
   - `sigprocmask()` → blokování signálu; signál bude zařazen do fronty či se jím bude řídit až jej process později odblokuje
   - `sigsuspend(sigmask)` → nastaví se blokované signály podle `sigmask` a process přejde do stavu *čekající* až do zaslání signálu, který není blokován/ignorován. Dle *[Košičan, prednaska07, slide 13]* není to samý, co `sigprocmask()` a `sleep()`, ???nechápu
2. Odpověď (Response) – **signál je přijat** a příslušná akce je vykonána
3. Vyčištění (Cleanup) – obnova normální operace procesu
   - Viz bod 3 předchozího rozfázování

Příklady scénářů synchronního a asynchronního signálu

Synchronní:

- výjimka (dělení nulou, nedovolená instrukce,...) způsobí přechod do módu jádro
- jádro vykoná její obsluhu a zašle se odpovídající signál běžícímu procesu
- při návratu z obsluhy proces najde signál

Asynchronní:

- uživatel stiskne **CTRL-C**

- generuje se přerušení (jako u každého stisknutí klávesy)
- ovladač rozpozná, že jde o kombinaci generující signál, a odešle signál **SIGINT** procesu v popředí
- když je proces naplánován jako běžící při návratu do uživatelského módu anebo byl-li běžící při návratu z přerušení, proces najde signál

Signály a vícevláknové procesy

Obsluha signálů v jednovláknových programech je přímočará; signály jsou vždy doručeny procesu. Doručení signálů je však komplikovanější u vícevláknových programů, kde proces může mít několik vláken. Kam se má potom signál doručit?

Obecně existují následující možnosti:

4. Doručit signál vláknu, ke kterému se signál vztahuje.
5. Doručit signál každému vláknu v procesu.
6. Doručit signál určitým vláknům v procesu.
7. Pověřit jedno konkrétní vlákno, aby přijímalo všechny signály pro proces.

Metoda pro doručení signálu záleží na typu generovaného signálu. Například synchronní signály je třeba doručit vláknu, které ten signál způsobilo a už ne ostatním vláknům procesu. U asynchronních signálů to ale není tak jasný. Některé asynchronní signály, třeba signál, který ukončuje proces (např. <control><C>), by měly být poslány všem vláknům.

Většina vícevláknových verzí UNIXu umožňuje vláknu určit, které signály bude přijímat a které blokovat. V některých případech proto může být asynchronní signál doručen pouze těm vláknům, která jej neblokují. Protože však signály musí být obslouženy pouze jednou, signál je obvykle doručen prvnímu nalezenému vláknu, které jej neblokuje.

Standardní UNIXová funkce pro doručení signálu je `kill (aid_t aid, int signal);` uvádíme zde proces (aid), kterému bude příslušný signál doručen. POSIX Pthreads taky ještě poskytují funkci `pthread_kill(pthread_t tid, int signal`, která umožňuje doručit signál konkrétnímu vláknu (tid.)

Windows APC

Ačkoliv Windows neposkytuje přímo podporu signálů, mohou být emulovány pomocí **asynchronních volání procedur** - ***asynchronous procedure calls*** **(APCs).** APC umožňuje uživatelskému vláknu (user thread) uvést funkci, která má být zavolána, když tomu user threadu přijde oznámení o určité události. Jak už název napovídá, APC je zhruba to samý co asynchronní signál v UNIXu. Zatímco se však UNIX musí potýkat s pořešením signálů ve vícevláknovém prostředí, možnost APC je přímočařejší, protože APC je doručeno konkrétnímu vláknu a ne procesu.

Příklady signálů

SIGINT – signál pro přerušení od terminálu (stisknutím Ctrl+C)

SIGQUIT – ukončí proces + core dump (záznam stavu pracovní paměti procesu do souboru, často při abnormálním ukončení)

SIGKILL – zabije proces, nemůže být zachycen ani ignorován + proces nemůže po jeho přijetí provést úklid

Proč může dojít ke zpoždění vyřízení signálu?

- signál je nevyřízen (*pending*), byl-li odeslán, ale nebyl přijat
- jenom jeden signál každého typu může být nevyřízen

Reakci na signál vykonává proces, kterému je signál zaslán, včetně ukončení procesu. To znamená, že musí být aspoň plánován stát se běžícím

Má-li nízkou prioritu, může mezi odesláním signálu a jeho přijetím, kdy se vykoná odpovídající akce, uplynout dosti dlouhá doba

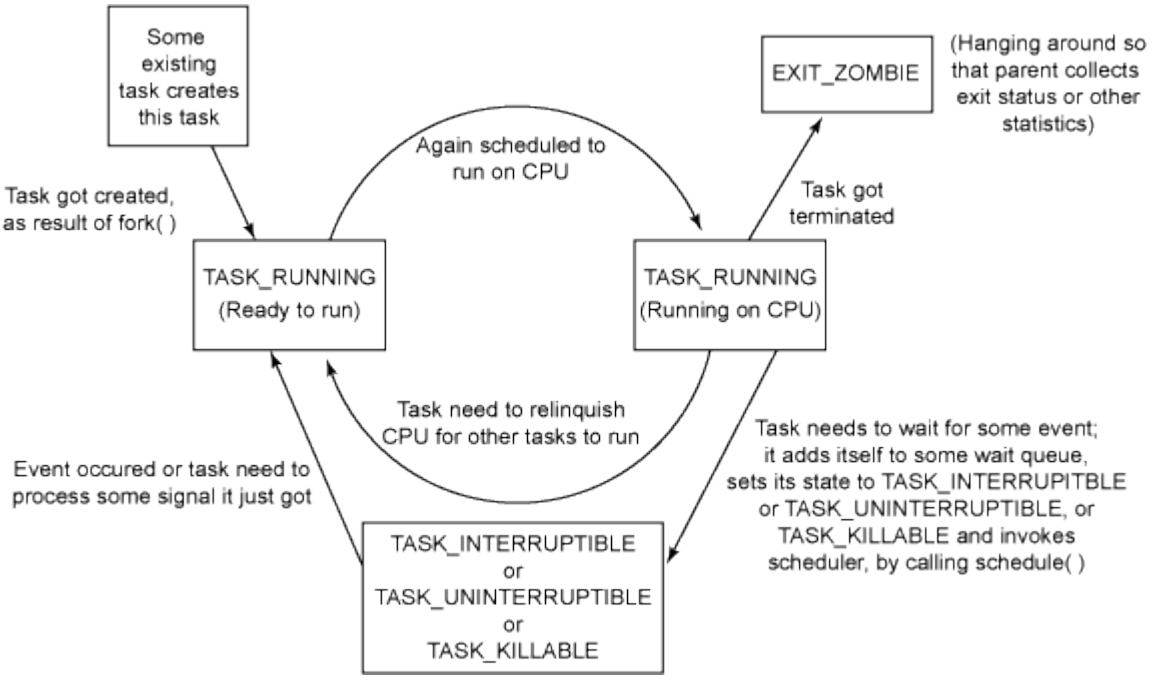
Další prodlení může způsobit, je-li proces v čase odeslání signálu ve stavu **zastaven** nebo **spící**

Co se má stát, když je odeslán signál spícímu procesu?

činnost jádra záleží na tom, proč proces přešel do stavu **spící**

- čeká-li **na událost, která zakrátko nastane**, např. dokončení diskové V/V operace, je spící v kategorii nepřerušitelný a signál je pouze zaznačen jako nevyřízen
- čeká-li **na událost, o které nevíme kdy nastane** nebo dokonce nemusí nastat vůbec, např. skončení potomka, vstup z terminálu, je spící v kategorii přerušitelný, je jádrem vzbuzen a přejde do stavu připraven

Linux nemá stav spící, ale stavy úloha_přerušitelná (***TASK_INTERRUPTIBLE***) a úloha_nepřerušitelná (***TASK_UNINTERRUPTIBLE***)



Spolehlivé a nespolehlivé signály

Nespolehlivé = můžou se ztratit; proces někdy zachytí signál, jindy ho ztratí; kvůli resetu signálu na implicitní akci.

Neumožňují ignorovat signál v daný moment, ale pamatovat si, že nastal, aby jej šlo blokovat a zpracovat později (třeba po skončení důležitého výpočtu).

Pokud to chceme emulovat (nastavit nějakou akci pro následující signál), musíme ji opět instalovat (= volat funkci `signal(sig, function)`) - vznik nedeterminismu, někdy se to stihne, někdy ne

Spolehlivé = perzistentní obslužné funkce signálů; blokování signálu, např. při obsluze signálů → nevznikne hnízdění

Další zdroje

- Pěkně vysvětleno na http://www.cs.uregina.ca/Links/class-info/330/Signals/signals.html

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Signály

signály umožňují oznámit procesům výskyt událostí v systému

jde o krátké zprávy, kde se procesům oznámí číslo signálu

systémová volání pro signály i vnitřní implementace se u jednotlivých variant a verzí značně liší, System V vs. BSD

problémy:

pro tvůrce přenositelných aplikací – může používat taková volání která jsou všude stejná

pro výrobce operačních systémů, které chtějí být kompatibilní s více variantami – musí poskytovat všechna systémová volání

standardní rozhraní specifikuje POSIX, včetně zpětné kompatibility

čísla některých signálů závisí na HW, označují se symbolickými konstantami SIG...

signály slouží dvěma hlavním účelům

- uvědomit proces, že nastala určitá událost
- přinutit proces vykonat funkci na zpracování signálu (*signal handler*)

systémová volání umožňují programátorovi zasílat signály a určit jak budou použity

POSIX

Signal	Default Action	Description
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped,
[XSI]		or continued.
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.
SIGUSR2	T	User-defined signal 2.
[XSI] SIGPOLL	T	Pollable event.
SIGPROF	T	Profiling timer expired.
SIGSYS	A	Bad system call.
SIGTRAP	A	Trace/breakpoint trap.
SIGURG	I	High bandwidth data is available at a socket.
[XSI] SIGVTALRM	T	Virtual timer expired.
SIGXCPU	A	CPU time limit exceeded.
SIGXFSZ	A	File size limit exceeded.

některé signály jsou vzhledem k procesu asynchronní
SIGINT, přerušení od terminálu stisknutím CTRL-C

jiné jsou synchronní
SIGSEV, chyba odkazu na stránku

jádro při zasílaní signálů rozlišuje dvě fáze:

- odeslání signálu
  jádro zaznamená v záznamu **`proc`** (deskriptoru procesu) zasílanému procesu odeslání nového signálu
- přijetí signálu
  jádro přinutí proces reagovat na signál

pro každý signál je nastavená implicitní reakce, která se vykoná, pokud ji proces nespecifikuje jinak

T (*abnormal*) *termination,* také *abort, exit*
proces je násilně ukončen se všemi důsledky volání **`exit(stav)`**, přitom **`stav`** indikuje pro **`wait()`** a **`waitpid()`** abnormální ukončení

A *abnormal termination,* také *dump, abort*
navíc se vykoná nějaká akce, typicky výpis obsahu pamětí procesu a hodnot registrů do souboru s názvem **core**

I *ignore*, signál je ignorovaný

S *stop*, proces je zastaven (*stopped*)

C *continue*, byl-li proces zastaven, může pokračovat, je převeden do stavu připraven, jinak je signál ignorován

proces může potlačit nastavenou akci a specifikovat jinou akci

- explicitně ignorovat signál
- zachytit signál a vykonat uživatelem definovanou funkci, která se nazývá, ošetření/obsluha signálu (*signal handler*)

na druhé straně, proces může obnovit reakci na signál na nastavenou implicitní akci

proces může signál blokovat, co znamená, že signál nebude přijat dokud signál není odblokován

signály SIGKILL a SIGSTOP nemůžou uživatelé ignorovat, blokovat nebo specifikovat pro ně obsluhu

signál je nevyřízen (*pending*), byl-li odeslán, ale nebyl přijat jenom jeden signál každého typu může být nevyřízen

reakci na signál vykonává proces, kterému je signál zaslán, včetně ukončení procesu, to znamená, že musí být aspoň plánován stát se běžícím

má-li nízkou prioritu může mezi odesláním signálu a jeho přijetím, kdy se vykoná odpovídající akce uplynout dosti dlouhá doba
další prodlení může způsobit je-li proces v čase odeslání signálu ve stavu

- **zastaven**
- **spící**

signály pro zastavení procesu (*stop signals*) **SIGSTOP, SIGTSTP, SIGTIN, SIGTOUT** mění okamžitě stav procesu na **zastaven** nebo **spící_a_zastaven** a signál **SIGCONT** je vrací do původního stavu

co se má stát, když je odeslán signál spícímu procesu?

činnost jádra záleží na tom proč proces přešel do stavu **spící**

- čeká-li na událost, která zakrátko nastane, např. dokončení diskové V/V operace, je spící v kategorii nepřerušitelný a signál je pouze zaznačen jako nevyřízen

- čeká-li na událost, o které nevíme kdy nastane nebo dokonce nemusí nastat vůbec, např. skončení potomka, vstup z terminálu, je spící v kategorii přerušitelný, je jádrem vzbuzen a přejde do stavu připraven

Linux nemá stav spící, ale stavy **úloha_přerušitelná** (***TASK_INTERRUPTIBLE***) **úloha_nepřerušitelná** (***TASK_UNINTERRUPTIBLE***)

přijímající proces je přinucen vykonat odpovídající akci, když pro něj jádro zavolá funkci **issig()** na zjištění nevyřízených signálů

jádro zavolá **issig()**:

- před návratem do uživatelského módu ze sytémového volání nebo z obsluhy přerušení
- před zablokováním procesu v přerušitelné kategorii
- když se stane běžícím po vzbuzení ze stavu spící v přerušitelné kategorii

když proces začal vykonávat systémové volání a nastane některý z posledních dvou případů, proces přijme signál a namísto dokončení systémového volání vykoná obsluhu signálu a systémové volání se obvykle vrátí s hodnotou **EINTR** v proměnné **errno**

scénář asynchronního signálu

- uživatel stiskne **CTRL-C**
- generuje se přerušení (jako u každého stisknutí klávesy)
- ovladač rozpozná, že jde o kombinaci generující signál a odešle signál **SIGINT** procesu v popředí
- když je proces naplánována jako běžící při návratu do uživatelského módu anebo byl-li běžící při návratu z přerušení proces najde signál

scénář synchronního signálu

- výjimka (dělení nulou, nedovolená instrukce,..) způsobí přechod do módu jádro
- jádro vykoná její obsluhu a zašle se odpovídající signál běžícímu procesu
- při návratu z obsluhy proces najde signál

Nespolehlivé signály

funkce pro obsluhu signálů nejsou perzistentní,
po zachycení (nalezení) signálu, jádro ještě před vyvoláním funkce obsluhy signálu nastaví implicitní akci, tedy pro následující signál, chceme-li opět vykonat obslužní funkci musíme ji znovu instalovat, vzniká soutěž (*race condition*)

instalace obslužní funkce signálu

```
oldfunction=signal(sig, function);
```

function
- **SIG_IGN** ignorování, ne **SIGKILL, SIGSTOP**
- **SIG_DFL** nastavit implicitní akci
- adresa obslužní funkce

sig číslo signálu

oldfunction předcházející obsluha

zaslání signálu

```
kill(pid,sig);
```

pid pid procesu, kterému bude zaslán signál (viz dále)

sig číslo signálu

Příklad

```
sig_obsluha()
{
   printf("signal zachycen");
   signal(SIGINT, sig_obsluha);
}

main()
{
   int rpid;

   signal(SIGINT, sig_obsluha);


   if (fork == 0)
   {
          sleep(5);
          rpid = getpid();
          for(;;)
             if (kill(rpid,SIGINT) == -1)
                   exit(1);
   }

   /* snížíme prioritu */
   nice(10);
   for(;;)
          ;
}
```

rodičovský proces má nízkou prioritu a je-li mu odebrán procesor v obslužné funkci **sig_obsluha()** signálu **SIGINT** před opětovnou instalací obslužní funkce a potomek zašle další signál, proces rodič při jeho přijetí vykoná nastavenou implicitní akci, tj. **exit**

bylo by řešením nenastavovat implicitní akci?

ano, ale při obsluze signálu by mohla být vnořena další obsluha, ... a uživatelský zásobník by mohl přetéct

Spolehlivé signály

- perzistentní obslužné funkce signálů
- blokování signálu, např. při obsluze signálů → nevznikne hnízdění

zaslání signálu

```
kill(pid, sig);
```

pid > 0 signál je zaslám procesu s PID = pid

pid = 0 signál je zaslán všem procesům skupiny

pid = -1 signál je zaslán všem procesům, kromě 0, 1 a běžícího

pid < -1 signál je zaslán všem procesům v skupině –pid

instalace obslužní funkce (náhrada signal)

```
sigaction(sig, act, oact);
```

specifikuje obsluhu pro signál **sig**

act ukazuje na záznam, který obsahuje:

- akci – **SIG_IGN, SIG_DFL**, nebo obslužní funkci
- masku signálů, které mají být blokovány při vykonávání obslužní funkce
- příznaky
  SA_NOCLDSTOP negeneruj **SIGCHLD**, když je potomek zastaven
  SA_RESTART signálem přerušené systémové volání, se restartuje
  SA_ONSTACK obslož signál v alternativním zásobníku deklarovaném voláním sigalstack()
  SA_RESETHAND akce se nastaví na implicitní
  SA_SIGINFO
  není-li nastaven obslužní funkce je zadána ve tvaru
  `func (sig);`
  je-li nastaven obslužní funkce je zadána ve tvaru
  `func (sig, info, kontext);`
  kde **info** vysvětluje příčinu vzniku signálu a **kontext** odkazuje na přerušený kontext procesu, když byl signál dodán
  SA_NOCLDWAIT nevytvářej mátohy, když potomci volajícího procesu skončí, zavolá-li proces **wait()** čeká až všichni potomci skončí
  SA_NODEFER neblokuj automaticky signál, když bude obsluhován, jako nespolehlivé signály

oact volitelně vrátí předcházející akci signálu

zjištění nevyřízených signálů

```
sigpending(set);
```

modifikování blokovaných signálů

```
sigprocmask(how, set, oset);
```

oset stará maska signálů

set nová maska signálů

how
SIG_BLOCK nová maska specifikuje signály, které se přidají k blokovaným
SIG_UNBLOCK nová maska specifikuje signály, kterých blokování se odstraní
SIG_SETMASK nová maska specifikuje blokované signály

čekání procesu na signál

`sigsuspend(sigmask);`

nastaví se blokované signály podle **`sigmask`** a proces přejde do stavu čekající až do zaslání signálu, který není blokován nebo ignorován

není ekvivalentní dvojici **`sigprocmask()`** a **`sleep()`** systémové volání **`sigprocmask()`** mohlo odblokovat signál, na který chceme čekat v **`sleep()`** a může se stát, že signál bude přijat před zavoláním **`sleep()`** a čekání nemusí skončit

obslužní funkce musí používat bezpečná (reentrantní) systémová volání

POSIX.1-2003

_Exit() _exit() abort() accept() access() aio_error() aio_return() aio_suspend() alarm() bind() cfgetispeed() cfgetospeed() cfsetispeed() cfsetospeed() chdir() chmod() chown() clock_gettime() close() connect() creat() dup() dup2() execle() execve() fchmod() fchown() fcntl() fdatasync() fork() fpathconf() fstat() fsync() ftruncate() getegid() geteuid() getgid() getgroups() getpeername() getpgrp() getpid() getppid() getsockname() getsockopt() getuid() kill() link() listen() lseek() lstat() mkdir() mkfifo() open() pathconf() pause() pipe() poll() posix_trace_event() pselect() raise() read() readlink() recv() recvfrom() recvmsg() rename() rmdir() select() sem_post() send() sendmsg() sendto() setgid() setpgid() setsid() setsockopt() setuid() shutdown() sigaction() sigaddset() sigdelset() sigemptyset() sigfillset() sigismember() signal() sigpause() sigpending() sigprocmask() sigqueue() sigset() sigsuspend() sleep() socket() socketpair() stat() symlink() sysconf() tcdrain() tcflow() tcflush() tcgetattr() tcgetpgrp() tcsendbreak() tcsetattr() tcsetpgrp() time() timer_getoverrun() timer_gettime() timer_settime() times() umask() uname() unlink() utime() wait() waitpid() write().

Implementace (Linux)

základní datová struktura pro uložení odeslaných signálů je pole bitů typu **`sigset_t`**, jeden bit pro každý signál

```
typedef struct {
   unsigned long sig[2];
} sigset_t;
```

`0` nemá žádný signál, v prvím prvku 31 tradičních signálů, ve druhém prvku signály pro reálný čas

deskriptor procesu obsahuje položky

`signal` typu **`sigset_t`** označující dodané signály

`blocked` typu **`sigset_t`** označující blokované signály

`sigpending` příznak, který je nastaven je-li jeden nebo více neblokovaných signálů nevyřízeno

`gsig` ukazatel na záznam **`signal_struct`** opisující obsluhu každého signálu

```
struct signal_struct {
   atomic_t             count;
   struct k_sigaction   action[64];
   spinlock_t           siglock;
};
```

`count` počet procesů (a vláken) sdílejících **`signal_struct`** – **`clone()`**, **`fork()`**, **`vfork()`**, **`CLONE_SIGHAND`** příznak nastaven

`siglock` zajišťuje výhradný přístup k položkám **`signal_struct`**

`action[64]` 64 **`k_sigaction`** záznamů specifikujících obsluhu jednotlivých signálů

- **`sa_handler`** – **`SIG_IGN`**, **`SIG_DFL`**, nebo ukazatel na obslužní funkci
- **`sa_flags`** – příznaky pro obsluhu signálu
- **`sa_mask`** – maskované signály při obsluze

Příklad1 – signal( )

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigobsluha()
{
  int pid,stav;
  pid = wait(&stav);
  /*exit ulozi navratovy kod v bitech 8 az 15*/
  printf("skoncil potomek %d s navratovym kodem
    %d\n",pid,stav/256);
}

main()
{
  signal(SIGCLD, sigobsluha); /*standardne je
    ignorovan*/

  if (fork() == 0)
    {
      printf("potomek pracuje\n");
      sleep(1);
      printf("potomek dopracoval\n");
      exit(1);
    };

  /*rodic neco dela*/
  printf("rodic pracuje\n");
  sleep(5);
  printf("rodic dopracoval\n");
  return(0);
}
```

Výstup:

```
potomek pracuje
rodic pracuje
potomek dopracoval
skoncil potomek 7734 s navratovym kodem 1
rodic dopracoval
```

Příklad2 – sigaction( )

```
#include <signal.h>
#include <stddef.h>
#include <stdio.h>
#include <sys/wait.h>

void zastav() {
  printf ("Nechci zastavit!\n");
}

main()
{
  int i;
  struct sigaction akce;
  sigset_t blokujvse;

  /*blokuj signaly, nechceme byt preruseni*/
  sigfillset (&blokujvse);
  akce.sa_mask = blokujvse;
  akce.sa_handler = zastav;
  akce.sa_flags = 0;

  sigaction (SIGTSTP, &akce, NULL);

  for (i=0; i<10; i++) {
       printf("Spim %d\n", i);
       sleep(2);
    }
}
```

Výstup:

```
Spim 0
Spim 1
                          CTRL Z
Nechci zastavit!
Spim 3
                          CTRL Z
Nechci zastavit!
...
```

Příklad3 – sigaction( ), siginfo

```
#include <stdio.h>
#include <signal.h>
#include <wait.h>
#include <ucontext.h>

void obsluha_potomka (int sig, siginfo_t *sip,
  void *notused)
{
  int stav;

  printf("Signal generoval proces: %d\n",
    sip->si_pid);
  fflush(stdout);

  /*WNOHANG neni-li skonceny potomek,
    waitpid() neceka a vrati 0*/
  if (sip->si_pid == waitpid(sip->si_pid,
       &stav, WNOHANG)){
    if(WIFEXITED(stav)){
       printf("Potomek skoncil, navratovy kod:
                %d.\n",WEXITSTATUS(stav));
    }
  else printf("Zadny potomek neskoncil\n");
  }
}
```

```
main()
{
  struct sigaction akce;

  akce.sa_sigaction = obsluha_potomka;
     /*ne sa_handler*/
  sigfillset(&akce.sa_mask);
  akce.sa_flags = SA_SIGINFO;
     /*jinak NULL*/

  sigaction(SIGCHLD, &akce, NULL);

  if (fork()== 0) {
    printf ("Potomek PID: %d\n", getpid());
    sleep(1);
  }
  else {
    printf ("Rodic PID: %d\n", getpid());
    sleep(5);
  };
}
```

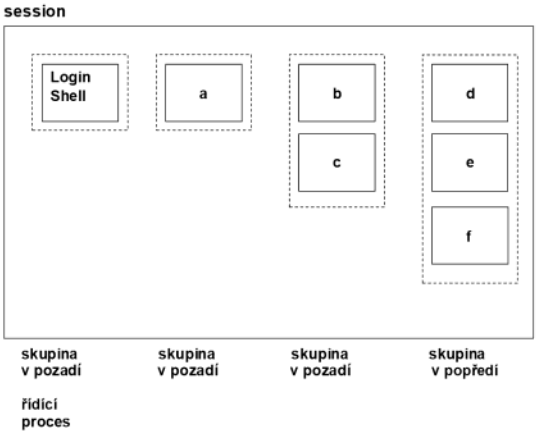
Výstup:

```
Potomek PID: 8502
Rodic PID: 8501
Potomek skoncil, navratovy kod: 0.
```

Session (práce, relace, sezení) a skupiny procesů

- umožňují vykonávat vícenásobné, souběžné úlohy (jobs) v jednom loginovém sezení, umístnit je do pozadí, přenést do popředí, zastavit je a umožnit pokračování.

```
$ a &
$ b | c &
$ d | e | f
$
```



- každý proces patří do skupiny procesů identifikované ukazatelem v deskriptoru procesu (**proc** záznamu)
- je vytvářen v průběhu **fork**
- procesy rodič, potomek, sourozenec jsou ve stejné skupině
- **PGID** je **PID** vedoucího procesu

po zahájení procesu ze shellu, je proces umístněn do vlastní skupiny voláním

```
int setpgid(pid, pgid);
```

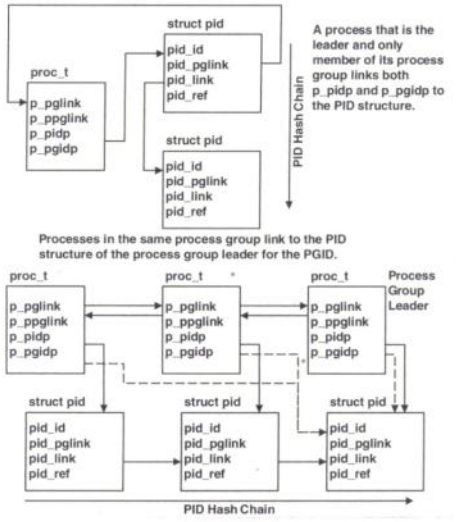
procesy kolony budou v jedné skupině, vedoucím bude první vytvořený proces

nejen shell, ale i aplikace mohou vytvářet skupiny procesů

signály možno zaslat všem procesům skupiny

skupina procesů může být v popředí nebo v pozadí
- procesy skupiny v popředí mají přístup k řídícímu (login) terminálu
- procesy skupiny v pozadí při čtení nebo zápisu na řídící terminál obdrží signál SIGTTIN nebo SIGTTOU

procesy jedné skupiny jsou v obousměrném spojovém seznamu, p_pglink a p_ppglink

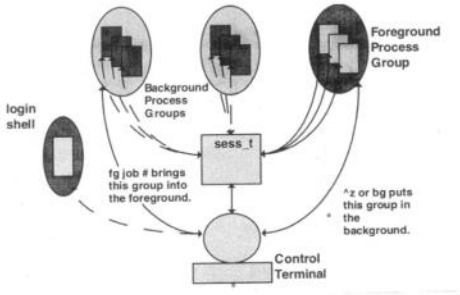


Zdroj: McDougallR., Mauro J.: Solaris Internals. Prentice Hall 2006

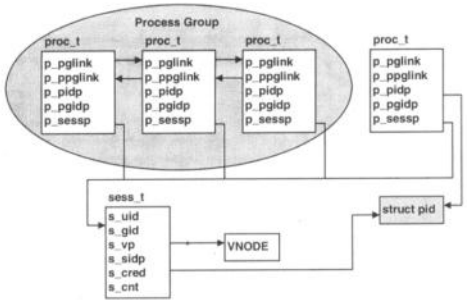
sezení obsahuje skupiny procesů se společným řídícím terminálem

- je reprezentováno datovou strukturou, na kterou ukazují procesy
- dědí se v průběhu **fork()**
- vedoucí sezení, proces který vytvořil spojení s řídícím terminálem, typicky login shell
- SID je PGID vedoucího skupiny

shell s řízením úloh
- po stisknutí CTRL Z vyšle všem procesům skupiny v popředí signál SIGTSTP a procesy jsou zastaveny a je možno je umístnit do pozadí - bg
- úlohu možno přenést do popředí příkazem fg, kdy vedoucí sezení voláním **tcsetpgrp()** jí přiřadí řídící terminál
-



Zdroj: McDougallR., Mauro J.: Solaris Internals. Prentice Hall 2006



Zdroj: McDougallR., Mauro J.: Solaris Internals. Prentice Hall 2006

Meziprocesová komunikace, roury, sdílená paměť, semafory, zasílaní zpráv - implementace.

Thursday, May 30, 2013 8:28 AM

Řada aplikací se skládá z mnoha spolupracujících procesů. Ty mezi sebou komunikuje a sdílejí informace.

Jádro OS musí poskytovat mechanismy, které to umožní – nazýváme je *prostředky meziprocesové komunikace*. Jejich **účelem je**:

- Přenos údajů
- Sdílení dat
- Oznámení vzniku událostí
- Sdílení prostředků
- Sledování a sdílení běhu procesu, např. při ladění programu

1. **Signály**
   - Umožňují oznámení procesům o asynchronní události
   - Viz otázka *„Signály“*
2. **Roury (pipes)**
   - Zápis dat na konec roury, čtení dat od začátku

     a. Nepojmenované roury

   - Vytvoření systémovým voláním **`pipe()`** – vrací dva deskriptory, jeden pro čtení, druhý pro zápis
   - Při vytváření procesů jsou deskriptory roury děděné
   - do roury může zapisovat i číst z ní více procesů, data jsou čtena v pořadí, v jakém byla zapsána
   - Procesy mohou **komunikovat** prostřednictvím roury, **pokud byla vytvořena společným předchůdcem**
   - Po skončení všech předchůdců přestává roura existovat

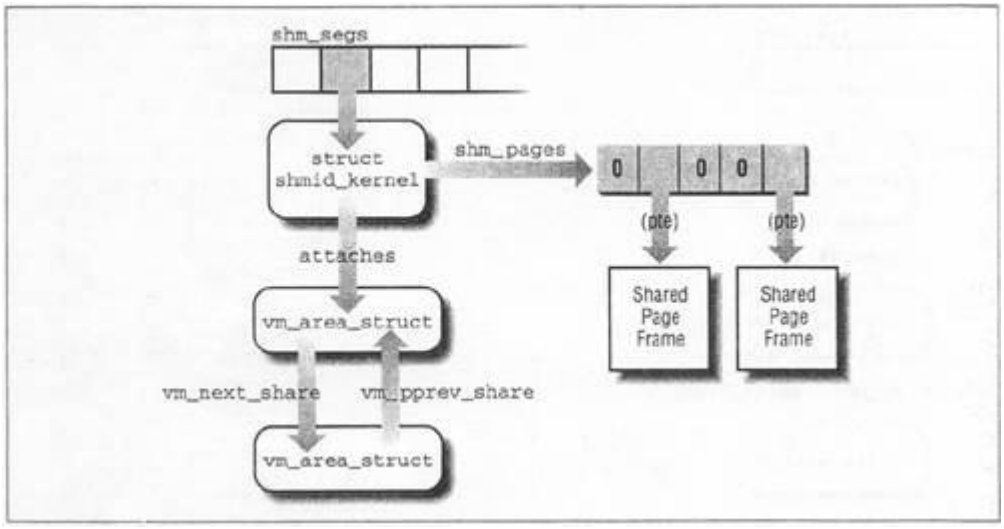
     b. Pojmenované roury, FIFO roury
   - **Perzistentní,** existují jako soubory, i když je nepoužívají žádné procesy
   - FIFO musí být explicitně zrušen, jako obyč soubory – **`unlink`**
   - Oproti obyč souborům jsou přečtená data odstraněna a z pohledu komunikace mají stejnou sémantiku jako nepojmenované roury
   - Vytvoření FIFO souboru: **`mknode(cesta, mód, zařízení)`**, **`mkfifo(cesta, mód)`**
     - Mód = obvyklá oprávnění
     - Zařízení = pro vytváření speciálních souborů pro zařízení
   - FIFO jsou pak otevřena sys voláním **`open()`** – vrací deskriptor souboru
   - do FIFO jde zapisovat sys voláním **`write()`** nebo z něj číst sys voláním **`read()`**

Následující způsoby mají podobnou implementaci – jsou identifikovány IPC (inter-process comunication) klíčem (obdoba cesty k souboru) a zpřístupňovány identifikátory IPC (obdoba deskriptoru souboru) – ty nejsou vázány na proces a nemění se v průběhu života objektu; získání identifikátoru IPC voláním `shmget(), semget(), msgget()`

3. **Sdílená paměť**
   - Oblast paměti, která je sdílená více procesy
   - Sdílená data jsou v paměti fyzicky uložena jen jednou, procesy je mají namapované do vlastního virtuálního paměťového prostoru (např. Používají DLL)
   - Proces ji **vytvoří/získá** voláním **`shmid = shmget(klíč, velikost, příznak);`**
     - Shared memory id, shared memory get
   - Proces **připojí oblast** na virtuální adresu voláním:
     `adr = shmat(shmid, shmadr, shmpříznak);`
     - „shared memory at“
     - `shmadr` je návrh adresy pro připojení oblasti; pokud je `shmadr` nula, jádro adresu

vybere
- skutečná adresa je návratová hodnota

- **Odpojení** oblasti:
  - `shmdt(shmaddr);`



4. **Semafory** - synchronizační primitivum s celočíselným čítačem a metodami P() a V()
   - **`semid = semget(klíč, počet, příznak)`**
     - vrátí identifikátor pole semaforů o velikost počet
   - **`stav = semop(semid, sops, nsops)`**
     - atomicky vykoná operace nad polem semaforů
     - `sops` je ukazatel na pole s `nsops` prvky typu `sembuf`

```
struct sembuf {
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
};
```

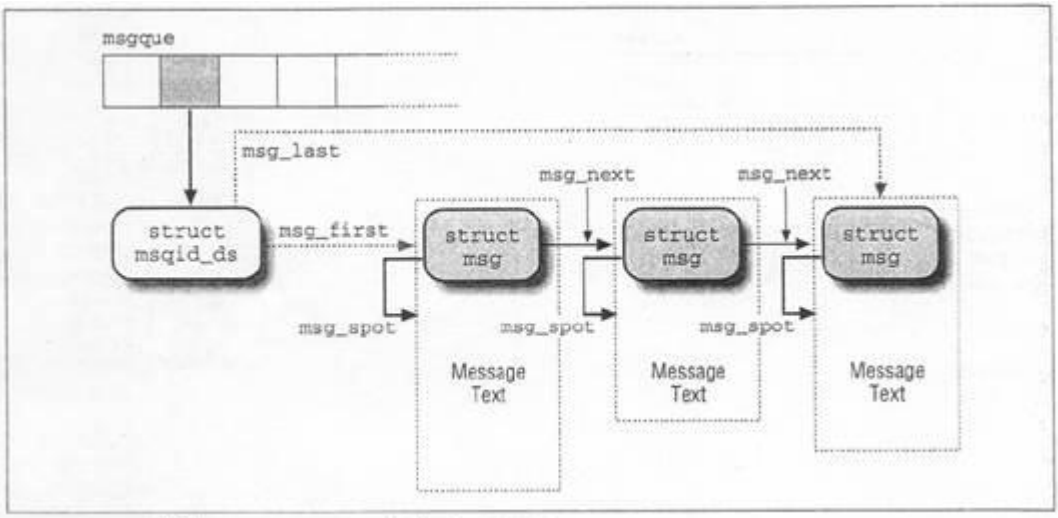
5. **Zasílání zpráv**
   - Procesy komunikují prostřednictvím zpráv
   - Zpráva vytvořená procesem je zaslána do fronty zpráv, je tam, dokud ji jiný proces nepřečte
   - Zpráva obsahuje 32bitový typ a data zprávy
   - Typ zprávy umožňuje selektivní výběr zpráv z fronty
   - Synchronní/asynchronní, blokující/neblokující
   - Proces **získá/vytvoří zprávu** voláním:
     - **`msgqid = msgget(klíč, příznak);`**
     - „message queue id“
   - Zpráva se **uloží do fronty** voláním:
     - **`msgsnd(msgqid, msgp, počet, příznak)`**
     - msgp = pointer na zprávu obsahující typ zprávy, který je následován
     - počet = velikost zprávy včetně typu v bytech
   - zprávy jsou ve frontě v pořadí jejich příchodu
   - **výběr zpráv** z fronty voláním:
     - **`počet = msgrcv(msgqid, msgp, maxpct, msgtyp, příznak)`**
     - je-li čtená zpráva delší než maxpct, je oseknutá
     - msgtyp = 0 → vrátí se první zpráva z fronty
     - msgtyp kladný → vrátí první zprávu typu msgtyp
     - msgtyp záporný → vrátí se první zpráva nejnižšího typu než je abs hodnota `msgtyp`

   - Na Windows:

     - SendMessage(HWND okna, int message, WPARAM. LPARAM) - blokující, čeká na vyzvednutí zprávy

     - SendNotifyMessage/PostMessage - neblokující

- Každé okno má svou smyčku zpráv while(GetMessage(..)) - blokující
- PeekMessage - neblokující



IPC message queue data structures

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Synchronizace v jádře, symetrický multiprocesor, atomické operace.

Thursday, May 30, 2013 8:29 AM

Problém kritické sekce - uvedení dat do nekonzistentního stavu nebo přerušení v okamžiku výpočtu

Kód jádrových funkcí se vykonává:

- Při systémovém volání
- Při obsluze výjimek a přerušení

Synchronizace v jádře

- Atomické operace
  - Atomická instrukce, jen jeden procesor aktivní v době jejího vykonání
- Spinlock
  - Klasický spinlock, u uniprocesoru řešen jen zákazem preempce jádra, uvnitř spinlocku je preempce jádra též zakázána, u multiprocesoru v čekací smyčce je preemplce jádra povolena a kernel tak místo toho může naplánovat jiný proces
- Read/Write spinlock
  - Oproti klasickému spinlocku zde readeři mohou číst neustále, writer musí ale čekat až všichni vše přečtou, pak to zamknout a zapsat; to samé u readerů, kteří všichni musejí čekat až jim to writer zas uvolní
- Seqlock

  When using read/write spin locks, requests issued by kernel control paths to perform a read_lock or a write_lock operation have the same priority: readers must wait until the writer has finished and, similarly, a writer must wait until all readers have finished.
  Seqlocks introduced in Linux 2.6 are similar to read/write spin locks, except that they give a much higher priority to writers:
  in fact a writer is allowed to proceed even when readers are active. The good part of this strategy is that a writer never waits (unless another writer is active);
  the bad part is that a reader may sometimes be forced to read the same data several times until it gets a valid copy.
  Each seqlock is a seqlock_t structure consisting of two fields:
  a lock field of type spinlock_t and an integer sequence field.
  This second field plays the role of a sequence counter. Each reader must read this sequence counter twice, before and after reading the data, and check whether the two values coincide.
  In the opposite case, a new writer has become active and has increased the sequence counter, thus implicitly telling the reader that the data just read is not valid.
- Kernel Semaphore

  However, whenever a kernel control path tries to acquire a busy resource protected by a kernel semaphore, the corresponding process is suspended. It becomes runnable again when the resource is released.
  Therefore, kernel semaphores can be acquired only by functions that are allowed to sleep:
  interrupt handlers and deferrable functions cannot use them.
- BKL (Big Kernel Lock)
- http://www.ibm.com/developerworks/linux/library/l-linux-synchronization/index.html

Systémová volání sdílejí jednotlivé struktury v jádře OS – snaha o zabránění soutěže nad systémovými daty prostřednictvím:

Nepreemptivnost procesů jádra

Jádrový proces nemůže být přerušen a nahrazen procesem s vyšší prioritou (pokud samotný proces neudělá yield). I tak ale může být přerušen výjimkou nebo přerušením, a obsluha přerušení může být zase přerušena přerušením.

Problém blokujících operací – možnost deadlocku -> uvolnit procesor před blokující operací

Atomické operace

Instrukce provádějící celou v jedné instrukci (inc, dec...)

Nemůže vzniknout nekonzistence dat – není přerušení

Pouze pro jednoprocesorový stroj, pro multiprocesorový je speciální instrukce zamykající sběrnici

Zákaz přerušení

Kritická oblast začíná zákazem přerušení a končí obnovením přerušení

KS musí být krátká a rychlá -> blokování I/O a procesoru

Velmi nebezpečné a nefunguje na víceprocesorových systémech

Nesmí se čekat na oznámení události přerušením

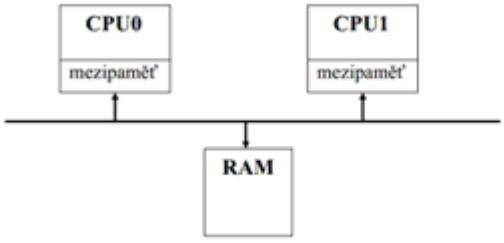
Zamykání

Pro víceprocesorové systémy velmi nákladné (režie)

Semafory v jádru, spinlock

Zabránění uvíznutí kdy více procesů žádá více zdrojů (přiřazení A->1, 2 B-> 2, 1) – prostředky se VŽDY alokují v pořadí adres semaforů

Symetrický multiprocesing SMP



- Procesory (identické) a sdílená hlavní pamět jsou připojeny ke společné sběrnici, tu je nutno zamykat pro výhradní přístup jednoho procesoru (aby nemohly 2 procesy zapisovat na stejné místo v paměti naráz).
- Klasické atomické instrukce (dec, inc) nejsou atomické, nutná speciální instrukce zamykající sběrnici
- Nutná synchronizace mezipaměti (cache) procesorů: když procesor modifikuje svou mezipamět, musí kontrolovat jestli stejná data nejsou v mezipaměti jiného procesoru a když ano, musí je aktualizovat nebo zneplatnit.
- Úloha může být zpracovávána postupně různými procesory, je umístěna ve sdílené hlavní paměti
- Na více procesorech naráz mohou být současně provedena systémová volání
- Obsluha přerušení – obsluhuje procesor, který má k dispozici nejvíce volných prostředků (obsluhuje

proces s nejnižší prioritou)

Nutno řešit pokročilým způsobem synchronizaci jádra – semafory + spinlock

spinlock – efektivní pro multiprocesory – blokující skončí na jiném procesoru a nemusí se nic přepínat

spinlock nesmí chránit data přístupná při obsluze přerušení - přerušení modifikuje data – problém řešený zákazem přerušení v KS

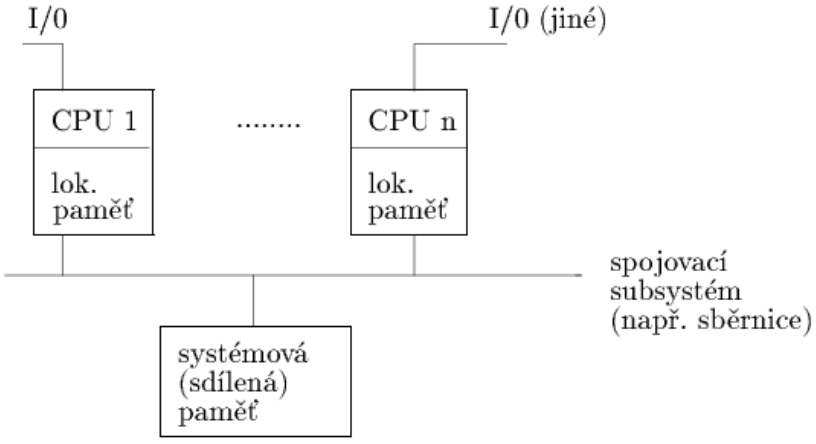
Procesy mohou využívat zámky dvěma způsoby:

sdílené - pro čtení (lepší propustnost)

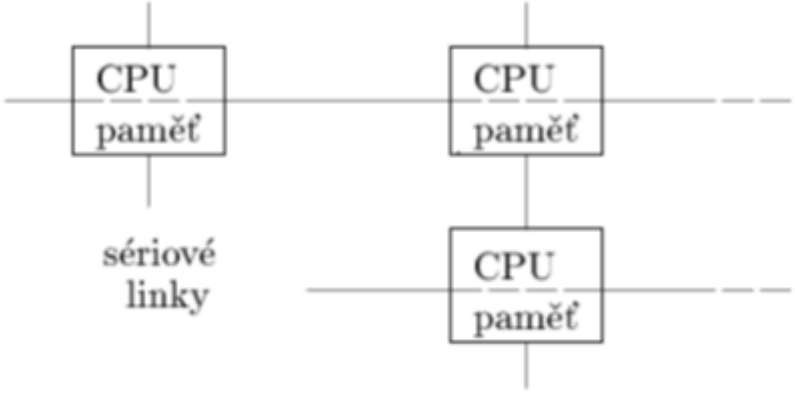
výhradní – pro zápis

Kromě SMP existují další technologie pracující na podobném principu:

- **asymetrické multiprocesory ASMP** - navíc vlastní lokální paměti a I/O připojení u procesorů, které tak mohou mít různé instrukční sady – každý procesor má speciální funkce – nelze libovolný proces na libovolný CPU



- **multiprocesor s distribuovanou pamětí** - procesory mají jen lokální paměti, není sdílená paměť, komunikace zasíláním zpráv, topologie 2D mřížky nebo N-rozměrné krychle, výhoda odstranění společné sběrnice jako úzkého hrdla



Atomické operace

- operace vykonávaná procesorem v jedné instrukci (inc, dec)
- problém u SMP – instrukce již není atomická v rámci procesů – nutno navíc uzamknout sběrnici instrukcí lock*
- atomická instrukce pro podporu semaforů a spinlocku – TSL (TestAndSetLock)

```
mov edx, DWORD(-1) //-1 zamčeno
//otestujeme stav zámku, 0 = odemčeno
spin: mov eax, [lockState]
test eax, eax
jnz spin
//zkusíme ho zamknout s -1
lock cmpxchg [lockState], edx
//nepředběhl nás jiný procesor?
//původní lockState je v eax
test eax, eax
jnz spin

Neoptimální verze:
while(!acquire_lock())
{ Sleep( 0 ); }
do_work();
release_lock();


spin_lock:
    TSL R, lock ;; atomicky provede R:=lock a lock:=1
    CMP R, 0 ;; byla v lock 0?
    JNE spin_lock ;; pokud nebyla (R<>0), byl zámek nastaven ->
    cyklus
    RET ;; návrat, tj. vstup do KS

spin_unlock:
    LD lock, 0 ;; ulož hodnotu 0 do lock
    RET
```

Z <https://d.docs.live.net/67cbdb2faf0647ea/Dokumenty/FAV/A11N0110P/Státní%20závěrečná%20zkouška/PPR/PPR_Karfik_v2.docx>

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Virtuální souborový systém, Extended File System.

Thursday, May 30, 2013 8:30 AM

http://www.ibm.com/developerworks/linux/library/l-virtual-filesystem-switch/index.html

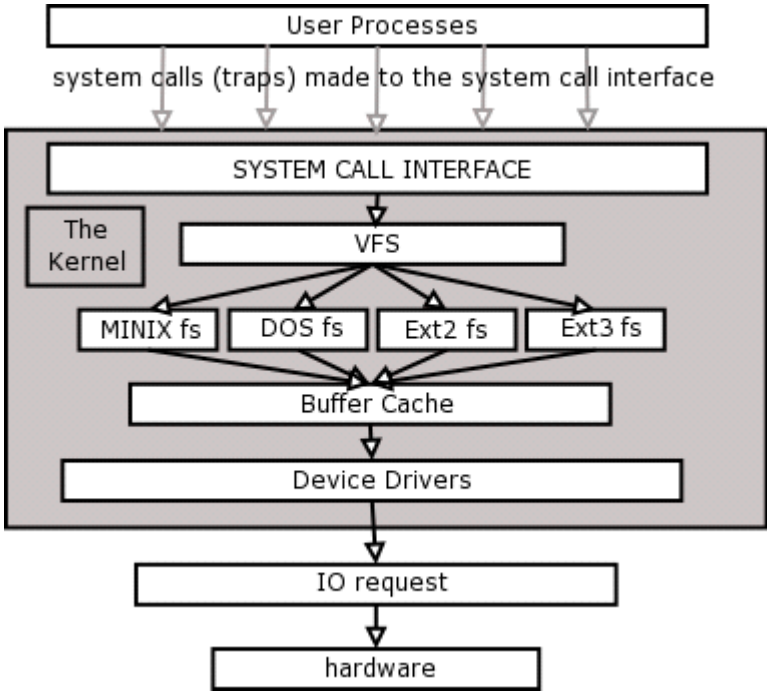
- Abstrakční vrstva(vrstva abstraktního API) mezi OS a konkrétním souborovým systémem.

- Na jedné straně je univerzální rozhraní pro operační systém a na druhé straně konkrétní ovladače pro všechny podporované souborové systémy

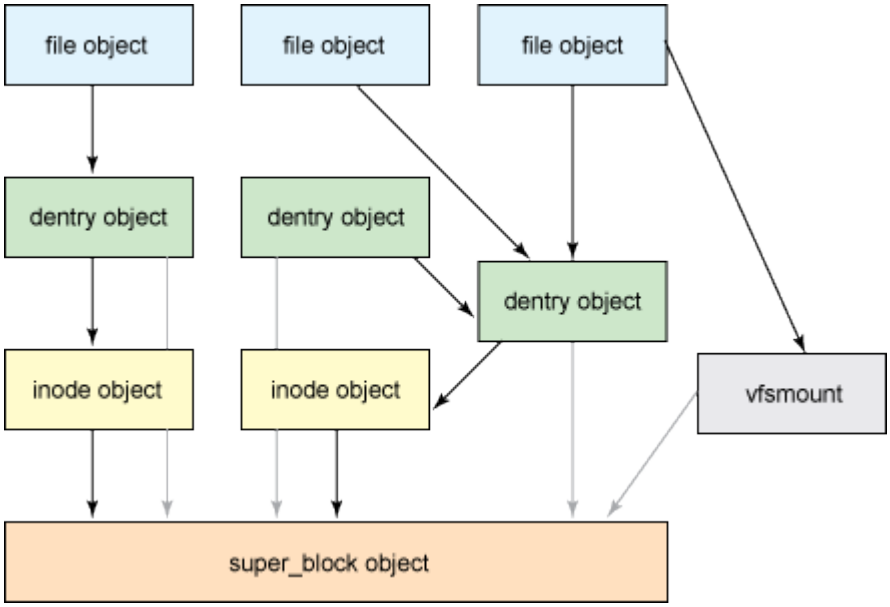
- Lze jednoduše přidávat podporu pro různé FS jenom novým ovladačem pro VFS bez potřeby upravovat jádro.

- Umožňuje zajistit dopřednou kompatibilitu OS s novými FS

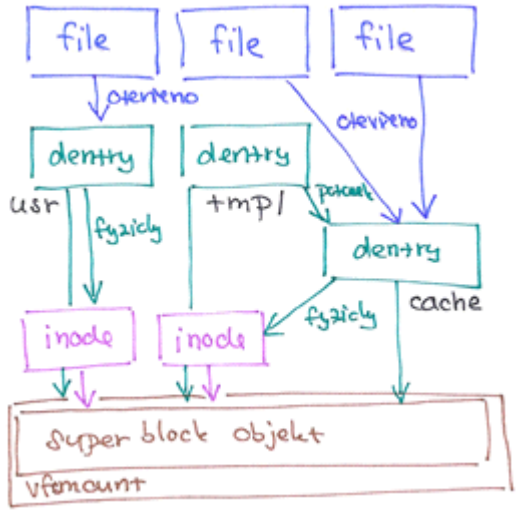
- rozhraní stejné pro všechny FS – aplikace nemusí znát podrobnosti o FS, jen využívá poskytnuté rozhraní



Struktura VFS Linux



From <http://www.bing.com/images/search?q=VFS&view=detail&id=0CE8CFF2EE03AC0EC78D8105CE040D3878E27A38&first=91&FORM=IDFRIR>



File

- struktura obsahující informace o otevřeném souboru (procesem)
- ukazatel na dentry = otevřený soubor (spojuje i-node – jeho číslo se jménem souboru)
- mód otevření, pozice v souboru, operace (read/write/seek)
- počet procesů sdílejících File (příkazem *dup*)

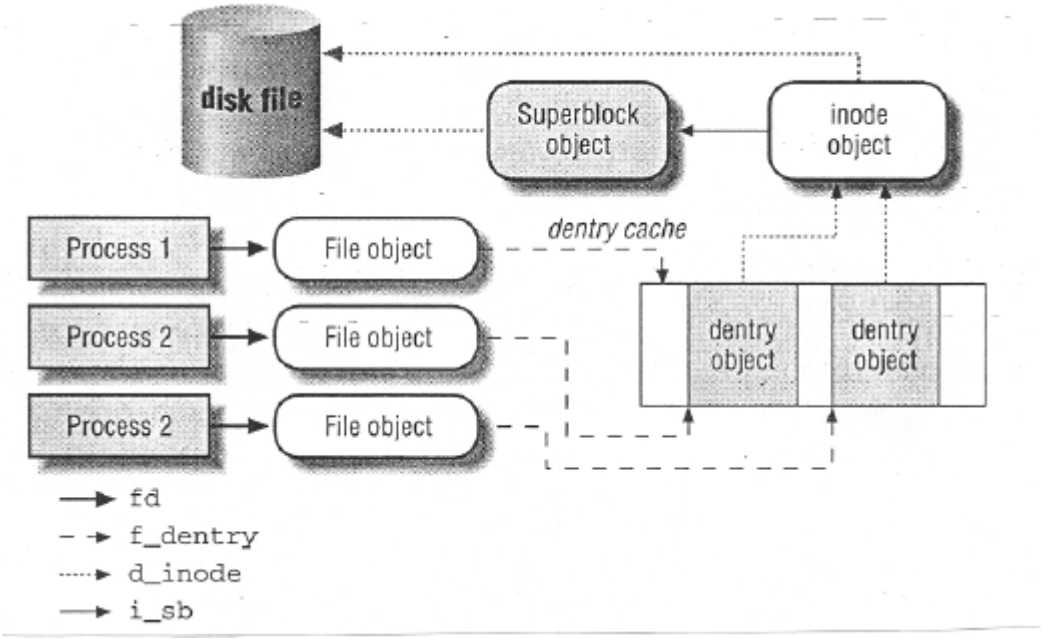
Dentry

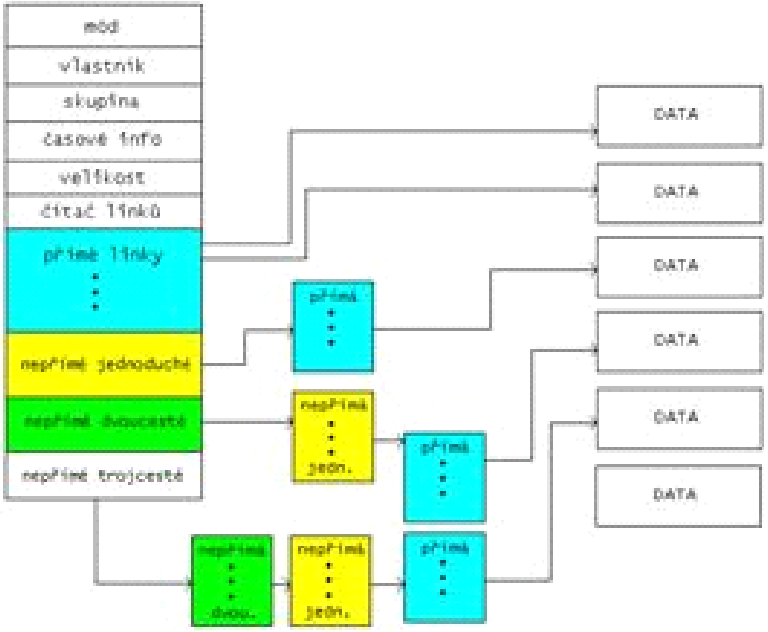
- položka hierarchické struktury, umožňuje vytvářet stromy v moderních FS
- rodič (root nemá, je jen jeden)
- potomci (adresáře + soubory)
- pouze virtuální (neexistuje na disku)
- sdružený inode objekt se samotnými daty a informacemi
- využití hash tabulky pro urychlení přístupu k položkám

Inode

- obsahuje veškeré informace pro VFS o souboru (souborem je i adresář, blokové zařízení)

    - Jen metadata, na datové bloky odkazuje
- typ souboru, vlastník, skupina, práva, ACL
- velikost souboru, data přidání, modifikace, mazání
- počet odkazů (soft/hard link)
- odkazy na i-uzly (jeho číslo) – přímé/nepřímé
- stav i-uzlu (dirty/clean)
- pro prohledávání je inode ve zřetězených seznamech: nepoužívané, používané, dirty
-





Superblok

- globální informace o fyzickém FS
- kořenový adresář, typ FS, seznamy i-uzlů
- všechny superbloky jsou ve spojovém seznamu

Připojení FS

- = registrace + připojení (začlenění do souborového systému)

- registrace zavedením modulu – OS má k dispozici funkce a informace pro připojení FS → body připojení = adresář, kde je FS připojen
- kořenový souborový systém = kořen souborového systému OS
- OS přečte superblock z disku a připojí kořenový adresář do cílového bodu připojení → původní adresář je zakrytý novým FS
- přidá do seznamu připojených souborových systémů

Odpojení FS:

- nelze odpojit kořenový FS
- nelze odpojit používaný FS (jeho soubory)
- odpojení způsobí flush na disk (zápis dirty souborů)

EXT (Extended File System)

- Vychází z UFS, navržen a vytvořen pro Linux
- EXT, EXT2 až EXT4
- Různé typy souborů: obyčený soubor, blokové a znakové zařízení
- Pevné odkazy, symbolické odkazy

Ext3

- žurnálovací (3 způsoby žurnálování)
- aktivně předchází fragemntaci (nelze jej defragmentovat když je připojený)
- bezpečnější mazání (složitější obnova smazaných souborů)
- zpětně kompatibilní s ext2

Ext4

- Zpětně kompatibilní s ext3 (fork a přidání funkčnosti) krom extentů
- Extent: nahrazení tradičního blokového rozdělení, zmenšuje fragmentaci a zlepšuje výkon při práci s velkými soubory (alokační jednotka o velikosti až 128 MB) – při použití nelze mountnout jako ext3
- Delayed allocation – alokování až při zápisu (zmenšuje fragmentaci)
- Rychlejší kontrola (přeskakování nealokovaného místa)
- Nanosekundový timestamp

FAT

Viz Kapitola/Záložka Bakalář

NTFS

http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Správa V/V zařízení.

Thursday, May 30, 2013 8:30 AM

Též I/O zařízení nebo periferie.
V/V (input/output) zařízení je hw zařízení které zprostředkuje kontakt počítače s okolím,

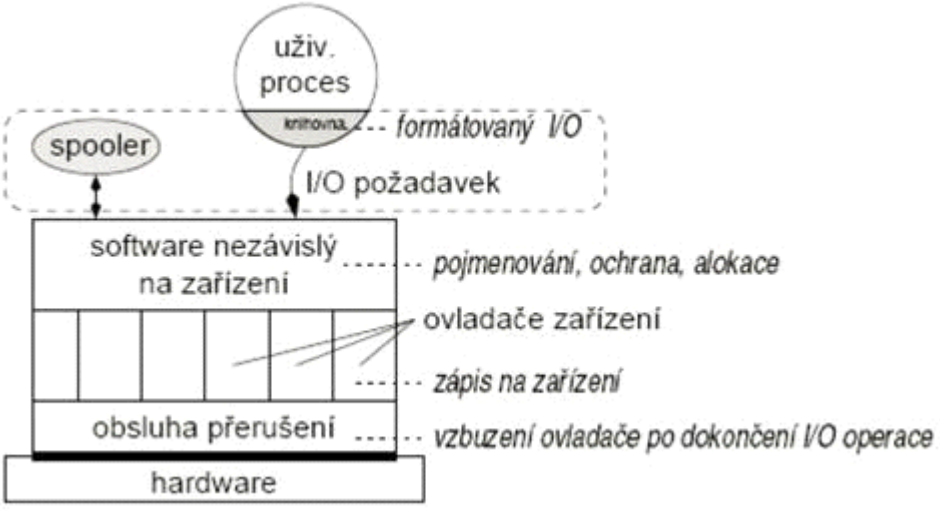
- Klávesnice/myš, čtečka kódů
- Obrazovka, tiskárna
  případně pseudo zařízení
- /dev/null
- /dev/random - generátor náhodných čísel

Principy I/O Software

- Všechny I/O instrukce jsou privilegované instrukce, a tedy probíhají v režimu jádra, operační system pak musí poskytovat prostředky pro provádění I/O

The I/O system se skládá z:

- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices



1.-3. – režim jádra:

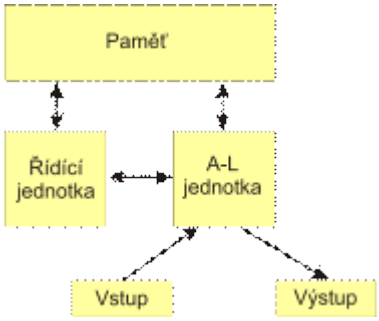
1. **Obsluha přerušení**
   - Řadič vyvolá přerušení ve chvíli **dokončení** I/O požadavku
   - Snaha, aby se přerušením nemusely zabývat vyšší vrstvy
   - Ovladač zadá I/O požadavek, **usne** (p(sem))
   - Po příchodu přerušení ho obsluha přerušení **vzbudí** (v)
   - Časově kritická obsluha přerušení – co nejkratší
2. **Ovladače zařízení**
   - Obsahují veškerý kód závislý na I/O zařízení = způsob komunikace s řadičem zařízení + zná detaily (ví o sektorech a stopách na disku, pohybech diskového raménka,…)
   - Ovládá všechna zařízení daného druhu nebo třídu podobných zařízení (ovladač SCSI disků → všechny SCSI disky
   - Ovladači je předán příkaz vyšší vrstvou (zapiš data do bloku n) → nový požadavek zařazen do fronty (může ještě být obsluhován předchozí) → ovladač zadá příkazy řadiči (až přijde požadavek na řadu; např. Přečtení sektoru) → zablokuje se do vykonávání požadavku (při rychlých operacích jako zápis do registru se neblokuje) → vzbuzení obsluhou přerušení (dokončení operace) + kontrola, zda nenastala chyba → pokud OK – předá výsledek (status + data) vyšší vrstvě → bere další požadavky ve frontě (1 vybere a spustí)
   - Ovladače často vytvářeny výrobci HW (dobře def. rozhraní mezi OS a ovladači)
   - Ovladače podobných zařízení – stejná rozhraní (síťové karty, zvukové karty,…)
3. **SW vrstva OS nezávislá na zařízení**

- Poskytuje I/O funkce společné pro všechna zařízení daného druhu (např. Společné fce pro všechna bloková zařízení)
- Definuje rozhraní s ovladači
- Poskytuje jednotné rozhraní uživatelskému SW
- **Funkce:**
  1. Pojmenování zařízení (LPT1 x /dev/lp0)
  2. Ochrana zařízení (přístupová práva)
  3. Alokace a uvolnění (v 1 chvíli použitelná jen 1 procesem – tiskárna, plotter, …)
  4. Vyrovnávací paměti (blok. zařízení – bloky pevné délky, pomalá zařízení – čtení/zápis s využitím bufferu)
  5. Hlášení chyb
  6. Jednotná velikost bloku pro bloková zařízení
- V Linuxu se zařízení jeví jako objekty v souborovém systému

4. **I/O SW v uživatelském režimu**
   - Programátor používá v programech **I/O funkce** nebo **příkazy jazyka**
   - `printf` v C, `writeln` v Pascalu | knihovny sestavené s programem | formátování – printf | často vlastní vyrovnávací paměť na jeden blok
   - **Spooling –** impl. pomocí procesů běžících v uživatelském režimu, = způsob obsluhy vyhrazených I/O zařízení
   - *příklad spoolingu*: přístup k tiskárně má pouze 1 speciální process – daemon lpd (má přístup do spooling directory, odkud vezme připravený soubor k vytisknutí, vytiskne ho a zruší)

Procesor komunikuje s V/V zařízeními pomocí registrů (můžou sloužit jako vyrovnávací paměť)

- **Izolované (port-mapped)**: Přístupné pomocí speciálních instrukcí
- **Vnitřní (IO mapped)**: namapovaná paměť (namapuju si CD-ROMku do paměti),
  - adresované jako paměť,
  - přístupné pomocí běžných paměťových instrukcí
  - např. DMA, velmi rychlé protože data nemusí do paměti přenášet procesor
  - Porušuje Von Neumannovu architekturu:



V/V zařízení si vyžádá obsluhu procesorem pomocí přerušení – 1 bitový kanál, který pouze upozorňuje procesor, že je třeba věnovat se V/V

Využití V/V zařízení v programu vyžaduje systémová volání (procesor musí běžet v privilegovaném režimu, do kterého se uživatelský program nesmí přepnout).

- Speciální konstrukce jazyka – např. proudy v C (stdio)

Soubor typu zařízení

Unixové OS mapují V/V zařízení do souborového systému (protože se snaží do soub. Systému mapovat úplně všechno)

Soubor se vytváří systémovým voláním **`mknod()`** s parametry

- Jméno
- Druh – blokové, znakové, roura

- Hlavní číslo – identifikuje skupinu (USB)
- Vedlejší číslo – konkrétní zařízení (třetí port zprava)

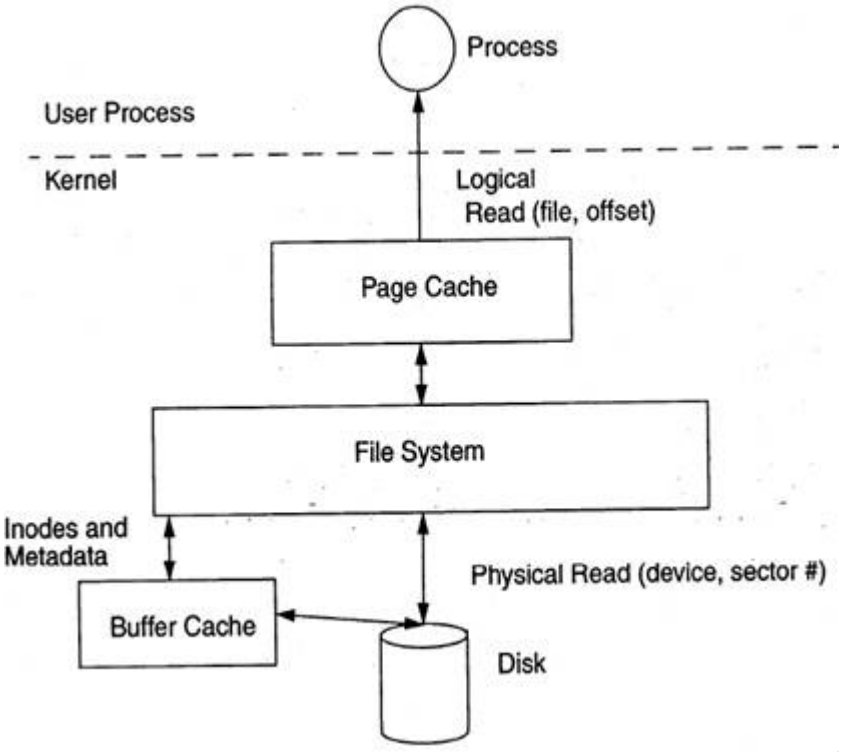
- **Bloková zařízení:** je potřeba zapisovat a číst vždy celý blok určené velikosti (video, zvuková karta)
  - **HDD**
  - Obsluhují se V/V operacemi s mezipamětí nebo stránkovými V/V operacemi
  - **Libovolný (random) přístup** k blokům zařízení
- **Znaková zařízení:** zapsán a čten je vždy jeden znak (terminál, COM a LPT porty)
  - **Klávesnice, myš**
  - Protože jde o jeden znak, nemají mezipaměť → data jsou přenášena primo do uživatelského adresového prostoru
  - Mechanizmus proudu – duplexní zpracování
  - **Sekvenční přístup** k datům

Síťová zařízení nemají svůj soubor.

Obsluha blokových zařízení

Přístup k blokovým zařízením musí být řádně plánován a synchronizován. K tomu účelu obvykle slouží modul jádra zvaný *I/O Scheduler* (*I/O plánovač*). Tento modul spravuje fronty požadavků na bloková zařízení a s použitím těchto front posílá požadavky ovladačům zařízení.

5. V/V operace s vyrovnávací pamětí
   - Ve vyrovnávací paměti je uložen diskový blok
   - Vyrovnávací paměti bloků jsou organizovány v mezipaměti vyrovnávacích pamětí bloků (*block buffer cache*) pomocí hlaviček bloků
6. Stránkové V/V operace
   - Data se přenášejí po blocích, adresový prostor procesu je množina stránek
   - V/V operace pro obyčejné soubory jsou vykonávány a ukládány po stránkách v mezipaměti stránek



[utlk, 629-725], [OSConcepts-7thEd, 549-580], [OSDesignAndImpl, 220-357]
[ZOS – pdf k 11a12 prednasce]

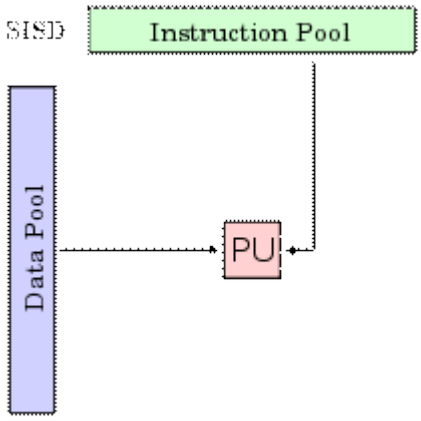
From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Modely SIMD, SPMD. MPSD, MPMD.

Wednesday, May 29, 2013 4:49 PM

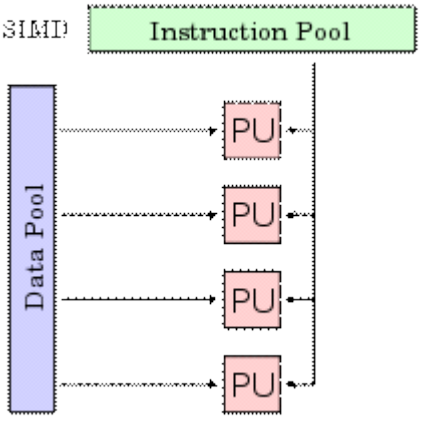
SISD (Single Instruction Single Data)

- Sekvenční výpočet, žádný paralelismus, např. MSDOS



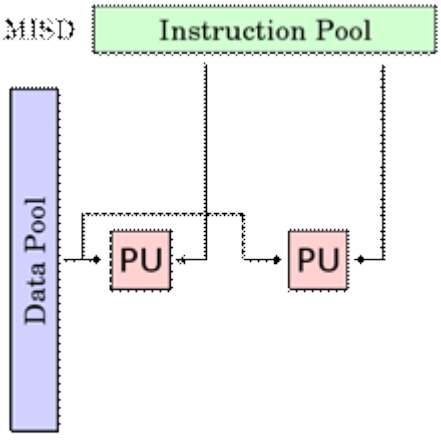
SIMD (Single Instruction Multiple Data)

- Datový paralelismus, např. vektorový paralelní počítač, maticové a vektorové výpočty na GPU (operace pro úpravu kontrastu v obrázku apod.), vektorová instrukce = jedna instrukce zpracuje několik dat najednou



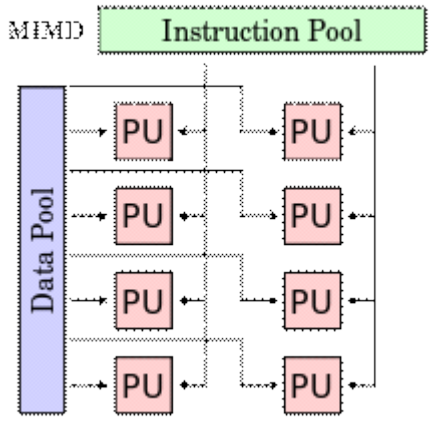
MISD (Multiple Instruction Single Data) = MPSD (Multiple Program Single Data)

- Používáno pro výpočty odolné proti poruchám, několik systémů zpracovává ta samá data a musejí se shodnout na výsledku – např. letadla. Používá se také v pipeline architektuře, kde několik procesů zpracovává data v jednom datovém proudu, např. montážní linka. Urychlení u pipeline je N (rovno počtu fází).



MIMD (Multiple Instruction Multiple Data)

Několik procesorů zároveň vykonává různé instrukce nad několika různými daty. Procesory pracují buď asynchronně, nebo mají sdílenou paměť – o zajištění integrity se stará OS, nebo distribuovanou paměť – má lepší škálovatelnost.



SPMD (Single Program Mutiple Data)

- Několik procesorů autonomně vykonává **jeden program nad různými daty**. Bod vykonávání programu nemusí být na všech procesorech stejný, označuje se též jako dekompozice dat.
- Používá se ke zpracování velkých objemů dat, k procesů běží podle stejného programu a zpracovává strukturou stejné, ale hodnotově různé části dat.
- Příklady použití: Monte Carlo, iterační numerická řešení

MPMD (Multiple Program Multiple Data)

- Několik procesorů autonomně vykonává **více než jeden program nad různými daty**. Např. farmer-worker, kdy jeden proces úkoluje ostatní. Nemusí jít nutně pouze o urychlení výpočtu – např. distribuované simulace, systém spolupracujících komponent.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Paralelizace cyklů.

Thursday, May 30, 2013 8:31 AM

Paralelizace cyklů

- Lze provést dekompozici dat, můžeme cykly paralelizovat a tím urychlit výpočet => datový paralelismus SPMD
- Příklady: nalezení minima/maxima v poli, součet prvků v poli

Rozdělení proměnných v paralelním výpočtu

- **Lokální proměnné** – inicializovány uvnitř smyčky
- **Sdílené proměnné**
  - **Nezávislé** – pokud jsou využívány jen pro čtení, nebo pokud v případě pole jde pouze o jeden prvek, se kterým je pracováno v jedné iteraci
  - **Závislé** – dále se dělí na:
    - **Redukční** – nejprve se čte a pak zapisuje (vše v jedné iteraci) – např. suma - stačí mi lokální kopie proměnné a dám to dohromady nakonec - viz concurrency runtime
    - **Uzamykané** – může být čtena i zapisována v několika iteracích (v rámci jedné iterace), nezáleží na pořadí iterací – např. max - musim pouzit zamky, jinak to nejde
    - **Uspořádané** – závisí na pořadí iterací - nepomuze mi nic (jen nejake chytre prevedeni na jiny problem, nelze paralelizovat)

Paralelizaci součtu pole lze provést snadno, protože se v cyklu vyskytuje nejvýše redukční proměnná, kterou stačí jen ošetřit mutexem.

Paralelizace cyklu se závislou uspořádanou proměnnou

S = urychlení = čas_neparalelizovaný/čas_paralelní

E = účinnost = urychlení/počet_cpu

Paralelizace sčítání prefixů (výsledek pole součtů) již nelze provést jednoduchým rozdělením iterací vláknům, protože se v cyklu nachází uspořádaná proměnná.

Uspořádaná proměnná nám brání zapisovat do pole v jiném než určeném pořadí. Proto zavedeme ještě jednu kopii pole. S i-1 pak přistupujeme do pole, do kterého se v cyklu již nezapisuje (zrušeno uspořádání). Poté můžeme přeuspořádat pořadí počítání prefixů do stromu s vrcholem uprostřed sčítané posloupnosti. Jestliže lze paralelizovat výpočet posledního prvku, lze paralelizovat i výpočet ostatních prvků. Pomocí úprav a optimalizací se můžeme dostat až na složitost O(n/m) u paralelní verze.

Loop Unrolling

- Nápověda pro překladač s autovektorizací
- V nejhorším dojde k většímu využití pipelines procesoru

- Loop Unrolling
  - Nápověda pro překladač s auto-vektorizací
  - V nejhorším dojde k většímu využití pipelines procesoru

Naivně	Lépe

`double a[100], sum;`
`int i;`
 	
`sum = 0.0f;`
 	
`for (i = 0;`
`  i < 100; i++) {`
`  sum += a[i];`
`}`
`//Překladač to může,`
`//ale i také nemusí`
`//správně pochopit.`
 	
`//Loop-unrolling`
`//také snižuje počet`
`//porovnávání, tj. i`
`//případných skoků.`
	`double a[100], sum;`
	`double sum1, sum2, sum3, sum4;`
	`int i;`
	
	`sum1 = 0.0f;`
	`sum2 = 0.0f;`
	`sum3 = 0.0f;`
	`sum4 = 0.0f;`
	
	`for (i = 0; i < 100; i + 4) {`
	`  sum1 += a[i];`
	`  sum2 += a[i+1];`
	`  sum3 += a[i+2];`
	`  sum4 += a[i+3];`
	`}`
	`sum = (sum4 + sum3) +`
	`  (sum1 + sum2);`

  - Použití i++ v a[…] by mohlo vnést závislost, tj. zhoršit výkon při využití pipelines

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Amdahlův a Gustafsonův zákon, Karp-Flattova metrika.

Tuesday, June 4, 2013 11:58 AM

Hodně dobrý materiál přímo od intelu, který popisuje nejen Amdahlův a Gustavsonův zákon, ale i jejich důsledky a důvody, proč dnes už Amdahl nestačí:
http://software.intel.com/sites/default/files/m/7/e/2/3/0/1638-Gillespie-0053-AAD_Gustafson-Amdahl_v1__2_.rh.final.pdf

Amdahlův zákon

$$E(p) = G + \frac{H}{p}$$
$$S(p) = \frac{(G+H)}{G+\frac{H}{p}}$$
$$S(p) \leq \frac{1}{f+\frac{1-f}{p}} \leq \frac{1}{f}$$ (f = sekvenčně prováděná část kódu)

Equation 1. A general representation of Amdahl's Law

$$Speedup\ (N) = \frac{1}{S + \frac{1 - S}{N}} - O_N$$

Where:
N = Number of processor cores
S = Serial percentage of workload (expressed as a decimal in the range 0-1)
O_N *= Parallelization overhead for N threads*

Předpokládá, že **velikost problému** zůstane při paralelizaci **stejná**

- Předpokládá tedy problém o konstantní velikosti
- Tzn. že s rostoucím počtem CPU se nemění rozsah úlohy
- Nevyužívá tak plně výpočetní sílu, která je k dispozici při navýšení počtu procesorů

Nelze dosáhnout perfektního urychlení, protože vždy bude nějaká část výpočtu provedena sekvenčně. G, H – čas strávený sekvenčně prováděným výpočtem. G je čas strávený vykonáváním neparalelizovatelného kódu. H je čas strávený výpočtem paralelizovatelného kódu, ale sekvenčně. Pro velké p platí S(p)=1 + H/G. To by znamenalo, že perfektní urychlení je možné, pokud se lze vyhnout kódu, který nelze paralelizovat. Ale čím větší počet procesorů, tím větší bude režie jejich komunikace, takže to přece jen nepůjde (režie OS je další faktor).

Nebere v potaz load balancing, režii komunikace

- Pro problém pevné velikosti s narůstajícím počtem CPU narůstá režie, což od jistého počtu CPU vede k pomalejšímu běhu než při menším počtu CPU

Anomalní urychlení.

Distribuce rozsáhlých dat u distribuované aplikace může omezit nutnost stránkovat RAM.
S dostatečně rychlými komunikačními kanály pak dojde k rychlejšímu vykonávání programového kódu, protože odpadá čekání na zpomalující IO operace provázející stránkování, včetně obsluhy příslušných přerušení. Např. paralelizované vyhledávací algoritmy mohou mít větší než lineární urychlení – lze rychleji upřesnit výběrová kritéria.

Superlinearní urychlení.

Je možné, aby S vyšlo lépe než perfektní urychlení. Nejedná se o chybu ve výpočtu, zejména, pokud porovnáváme proti nejlepšímu sekvenčnímu algoritmu. Vliv na to má cache procesoru, především případ, kdy nastane mnohem častější cache hit než je obvyklé. Výpočet je pak prováděn mnohem rychleji, než když se programový kód dostává k procesoru z pomalejší paměti. Záleží na konkrétním programovém kódu, zda se ho bude dostatečné množství nalézat právě v lokální cache jednotlivých

procesorů. Analogicky to platí i pro datovou cache.

Gustavsonův zákon

Říká, že výpočty nad velkou množinou dat mohou být efektivně paralelizovány.

Změna oproti Amdahlovi - místo konstatního rozsahu **se uvažuje konstantní doba běhu**.

- Pokud jde něco paralelizovat, vyřeší se za stejnou dobu větší problémy ("vypočte se toho víc")

Pokud bude sériová část zanedbatelná a budeme mít k dispozici spoustu procesoru s distribuovanou paměti, Amdahl nám nepomůže. A(n)=čas výpočtu sériově, b(n) = čas výpočtu paralelně na p procesorech.

$a(n) + b(n) = 1$ výpočet na p CPU

$a(n) + p * b(n) = 1$ počet na jednom CPU

$S = \frac{a(n)+p*b(n)}{a(n)+b(n)}$

Equation 3. A computational representation of Gustafson's Trend

$$Speedup\ (N) = \frac{S + N\ (1 - S)}{S + (1 - S)} - O_N$$

Where:
N = Number of processor cores
S = Serial percentage of workload (expressed as a decimal in the range 0-1)
O_N *= Parallelization overhead for N threads*

Karp-Flattova metrika

$$e = \frac{\frac{1}{y} - \frac{1}{p}}{1 - \frac{1}{p}} =$$

určuje podíl, kolik kódu se provedlo sériově - y (má to být psí) určuje urychlení na p procesorech.

- Např. z tabulky s počtem využitých procesorů (první řádka) a dosaženého urychlení (druhá řádka) lze vypočítat pro jednotlivé počty CPU sériově prováděný podíl - *e*
    - Pokud *e* zůstává stejné s přibývajícím počtem CPU - hlavním důvodem je omezená možnost paralelismu (a urychlení *psí* narůstá čím dál méně)
    - Pokud *e* s přibývajícím počtem CPU roste, důvodem k slabému urychlení je režie paralelismu (buď se týká nastartování procesů, komunikace, synchronizace nebo omezení architektury)
    - Pokud *e* postupně klesá = ideál, čím dál větší část výpočtu se provádí paralelně

Faktory ovlivňující rychlost vykonávání programového kódu.

Thursday, May 30, 2013 8:32 AM

Motivace

• State of the Art schopností procesorů

- Superskalární architektura
- CISC instrukce se překládají na mikrokód, aby i procesory s CISCovou instrukční sadou mohly těžit z výhod RISCové architektury
- Pipelining
- Vykonávání instrukcí mimo pořadí
- Přejmenovávání registrů
- Spekulativní vykonávání kódu dopředu
- Odhadování výsledků podmínek skoků
- Vektorové instrukce

Ve výsledku může jedno jádro procesoru vykonávat celé instrukce, nebo alespoň části instrukcí, paralelně

- Tj. co programátor píše jako kód jednoho vlákna, se ve skutečnosti vykonává paralelně
- Tj. uvedené optimalizace spadají do rámce PPR
  - Více vláken = principiální pohled na PPR
  - +Optimalizace = reakce na aktuální stav
  - Co bývalo výhodné paralelizovat, to dnes může rychleji spočítat „sériový kód“
- Dobrý překladač může vytvořit takový strojový kód, který umí využít paralelizačních schopností procesoru
  - Proto se liší výkonnost kódu v závislosti na překladači a zvolené míře optimalizace
  - Ale ani ten nejlepší překladač neumí odhadnout vše
- Proto je třeba mu umět pomoci takovým zápisem kódu, ze kterého toho pozná co nejvíce
  - A proto je třeba znát, jak to vypadá z pohledu procesoru
  - Low-level techniky, pokud pro to není zvláštní důvod, se vyplatí nechat na překladači
  - Ale algoritmy na vyšší úrovni jsou záležitosti pro programátora

Základní myšlenky

- Co lze, to se vypočítá pouze jednou
- Co lze, to se vypočítá paralelně
  - Je však třeba zvolit správnou granularitu výpočtu
  - Paralelizace totiž není vždy výhodná
- V některých případech je sériový kód rychlejší, má menší režii – nevytváří a nesynchronizuje vlákna
- Paměť se alokuje co nejméně, využívá se již alokovaná paměť
- Program se píše tak, aby operační systém co nejméně swappoval
- Redukovat náklady na vytvoření a synchronizaci vláken
- Běžet co nejvíce v uživatelském adresovém prostoru
- Ale hlavně, psát efektivní kód už samotného vlákna
- Paralelizovaný neefektivní kód bude s největší pravděpodobností pomalejší než efektivně napsaný sériový kód

Skoky

- Eliminace skoků má zásadní vliv na zvýšení výkonu (jump je hodně náročnej)
- Využit instrukce cmov – bytecode pro to nemá ekvivalent
  - Cílem je uspořádat podmínky, aby si procesor správně tipnul, která instrukce bude další
- Při adresování paměti přes pointery *m musí překladač předpokládat, že *m může ukazovat kamkoli a nemá moc možností, jak optimalizovat. Proto vzhledem k optimalizacím je dobré použít zápis přes indexy, kdy takový zápis obsahuje jakousi nápovědu pro překladač, jak to celé optimalizovat.

Eliminace nepotřebných sekvencí Store-Load

- Používání dočasných proměnných, které mohou být realizovány s pomocí registru
- Programátor by měl pomoci překladači jejich použitím – namísto co nejkratšího zápisu kódu

Konverze operandů

- Nemíchat operandy různé velikosti, konverze to zpomaluje

Loop Unrolling

- Nápověda pro překladač s autovektorizací
- V nejhorším dojde k většímu využití pipelines procesoru

- Loop Unrolling
  - Nápověda pro překladač s auto-vektorizací
  - V nejhorším dojde k většímu využití pipelines procesoru

Naivně	Lépe

`double a[100], sum;`
`int i;`
 	
`sum = 0.0f;`
 	
`for (i = 0;`
`  i < 100; i++) {`
`  sum += a[i];`
`}`
`//Překladač to může,`
`//ale i také nemusí`
`//správně pochopit.`
 	
`//Loop-unrolling`
`//také snižuje počet`
`//porovnávání, tj. i`
`//případných skoků.`
	`double a[100], sum;`
	`double sum1, sum2, sum3, sum4;`
	`int i;`
	
	`sum1 = 0.0f;`
	`sum2 = 0.0f;`
	`sum3 = 0.0f;`
	`sum4 = 0.0f;`
	
	`for (i = 0; i < 100; i + 4) {`
	`  sum1 += a[i];`
	`  sum2 += a[i+1];`
	`  sum3 += a[i+2];`
	`  sum4 += a[i+3];`
	`}`
	`sum = (sum4 + sum3) +`
	`  (sum1 + sum2);`

  - Použití i++ v a[…] by mohlo vnést závislost, tj. zhoršit výkon při využití pipelines

Vyhodnocování podmínek

- Podmínky jsou obvykle vyhodnocovány ve zkrácené formě – cílem je seřadit podmínky tak, aby při jejich vyhodnocování bylo zapotřebí provést co nejméně úkonů, tzn. Bylo co nejrychleji vyřazeno co nejvíce možností a aby byly co nejmenší nároky na vyhodnocení podmínky.
- Sekvence AND končí prvním FALSE, sekvence OR končí prvním TRUE.

Branch prediction – předpovídání skoků

- procesor obsahuje tzv. Branch prediction, kdy spekulativně vykonává kód dopředu podle toho, jaký předpokládá výsledek skoku. Uvedená konverze nemění stav branch-prediction, tj. Neovlivňuje urychlování volající funkce, a kdyby se alokovala paměť, např. Pro nový string, tak už to stav branch-prediction ovlivní.

Kopírování paměti

- kopírování celého bloku paměti místo prvek po prvku

Garbage Collector

- způsobuje náhlé, nedeterministické vytěžování systému

Synchronizace

- jinak než kritickou sekcí, např. Pomocí InterlockedIncrement apod. Kritická sekce má velkou režii.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> fq

Programové prostředky pro multithreading - Java, POSIX, WinAPI, OpenMP.

Thursday, May 30, 2013 8:32 AM

Java

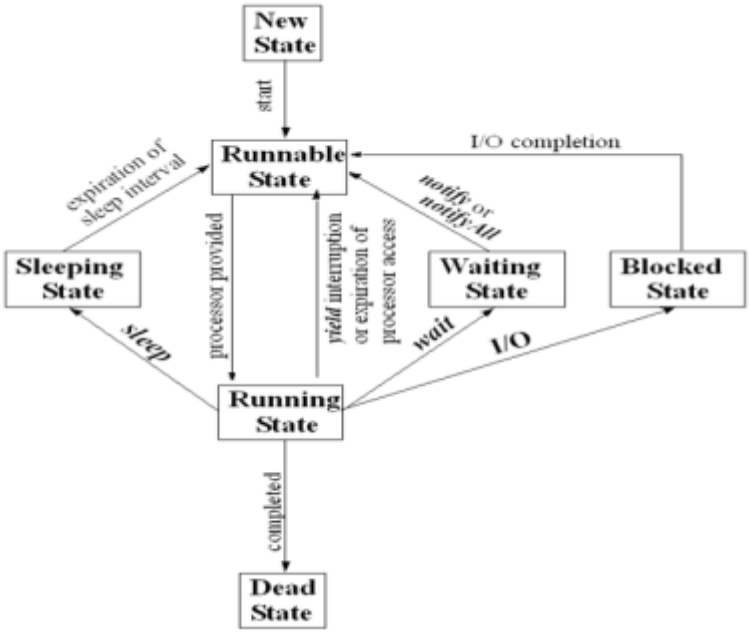
- V programovacím jazyce Java jsou objekty a metody potřebné pro práci s vlákny dostupné ve standardních knihovnách.
- Základem je třída Thread

Thread

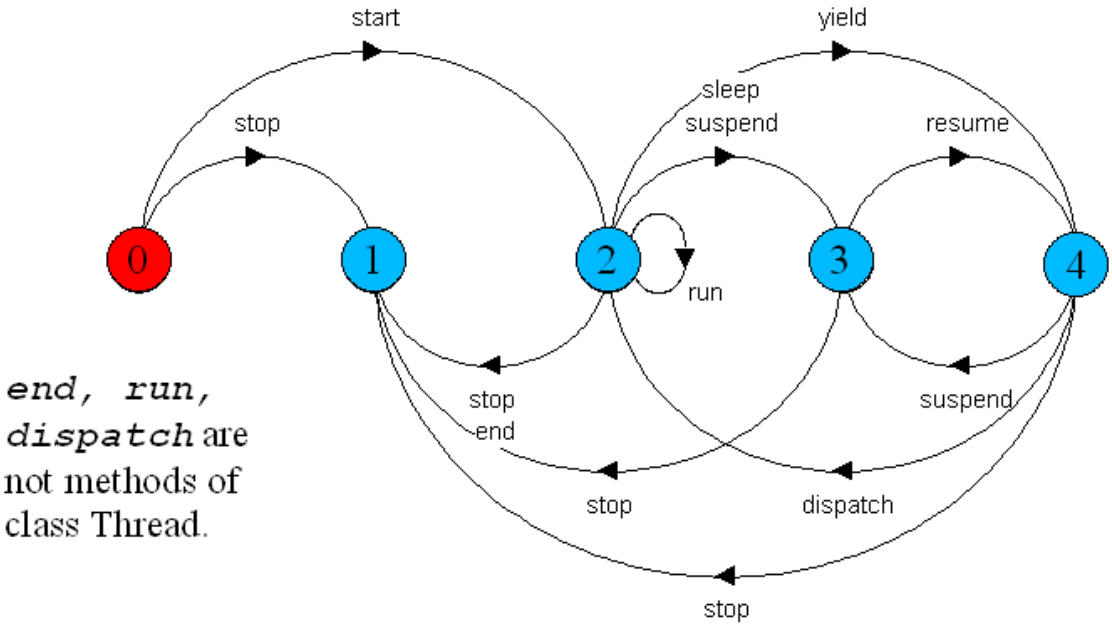
- tvoří základ všech paralelních programů v Javě
- pro jednoduché programy stačí oddědit od této třídy a překrýt metodu *run()*, do které se napíše výkonný kód vlákna.
- protože někdy je potřeba, aby naše třída dědila od jiné a zároveň měla vlastnosti vlákna, existuje ještě rozhraní *Runnable*, které stačí implementovat a třída rovněž získá vlastnosti vlákna
- pro napsání paralelního programu stačí vytvořit potřebné třídy, napsat jejich výkonné kódy do metod *run(),* a pak vlákna spustit (vlákna se spouští metodou *start()*

- *Stop, resume, suspend* - ***deprecated***
    - Stop - po jejím volání vlákno uvolní všechny držené monitory --> může zanechat uvolněné objekty v nekonzistentním stavu

Stavy vlákna



V javě:



States 0 to 4 correspond to **CREATED**, **TERMINATED**, **RUNNING**, **NON-RUNNABLE**, and **RUNNABLE** respectively.

Monitory v Javě

- komunikace vláken je řešena přes sdílenou paměť
- kritické sekce jsou ošetřeny klíčovým slovem ***synchronized***
- monitor je součástí každého objektu
- pro implementaci monitorů jsou uvnitř objektu monitoru skryté atributy:
  - **1 zámek** – pro všechny synchronizované metody
  - **1 fronta** – pro blokovaná vlákna čekající na vstup do synchronizované metody (kritická sekce)
  - nad frontou fungují privátní metody monitoru (objektu) *wait()*, *notify()* a *notifyAll()*
  - kromě těchto implicitních monitorů objektu lze v metodě ještě vytvořit synchronizační blok, který může použít i jiný zámek, než je implicitní zámek objektu, což dovoluje jemnější členění vzájemného vyloučení (např. jen na část metody)

POSIX

Struktury v posixu jsou reprezentovány pomocí *handle*. Struktury jsou **pthread_t, pthreat_mutex_t** a **pthread_cond_t**. Objekty mají sadu atributů, které lze měnit pthread_attr_t, pthread_mutexattr_t, pthread_condattr_t. Funkce pro manipulaci s objekty buď vracejou 0 jako OK, nebo jinou hodnotu jako chybový kód.

Vlákna

Program vlákna je vlastně funkce C s typem void* prog_name(void* arg)

vlákno se vytvoří následujícím způsobem:

```
int status; //0 uspech, jinak chyba
status = pthread_create(&worker, NULL, prog_name,…);
```

- vlákno běží hned po vytvoření
- stavy vlákna jsou **ready, running, waiting** a **terminated**
- vlákno se ruší pomocí *pthread_exit* (ze stejného vlákna) nebo *pthread_cancel* (z cizího)
- Atributy:
  - **způsob plánování**: FIFO – nejprioritnější kategorie (nejdříve se plánují vlákna tétokategorie), RR – round-robin, FG – foreground, implicitní kategorie, střídání vláken, ty s vyšší prioritou mají více času, BG – background, všechna vlákna se střídají, ale dostávají méně času než FG
  - **priorita**
  - **rozměr zásobníku**
  - **hlídač zásobníku** – jak daleko se můžeme od konce zásobníku dostat, jinak vznikne výjimka
  - **konec vlákna** - vlákno dojde na konec svého programu - na toto ukončení se lze

synchronizovat z jiného vlákna pomocí pthread_join(na_koho_se_čeká, kam_přijde_výsledek) nebo zabito z vnějšku – pokud možno nepoužívat, základní funkce pro likvidaci pthread_cancel(oběť); oběť se může bránit pthread_setcancelstate(…), k likvidaci nemůže dojít kdekoliv v kódu vlákna, ale jen v předem připravených místech (volání blokující funkce, volání pthread_testcancel()), popisovaný způsob je synchronní, existuje i asynchronní)

Mutex

Je to synchronizační primitivum, umožňuje implementaci vzájemného vyloučení v kritické sekci. Jen jedno vlákno vlastnící mutex může být v kritické sekci, ostatní vlákna čekají.

```
Pthread_mutex_lock(mutex_handle)
Pthread_mutex_unlock(mutex_handle)
State = pthread_try_lock(mutex_handle)
```

Podmínková proměnná

Vlákno se uspí do té doby, dokud se nesplní nějaká podmínka, která se pak signalizuje pomocí podmínkové proměnné. Podmínková proměnná je svázána s mutexem.

```
Pthread_cond_init
Pthread_cond_wait - blokující operace, vlákno se převede do stavu waiting, a odemkne se zámek
Pthread_cond_timedwait
Pthread_cond_signal
Pthread_cond_broadcast - jako notifyAll()
Pthread_cond_destroy
```

WINAPI

Process

- Má virtuální paměťový prostor, systémové zdroje, bezpečnostní kontext (kdokoliv nemůže provádět cokoliv), jedinečný identifikátor, spustitelný kód
- Prioritní třída – vlákna pak mohou mít prioritu pouze v rozsahu prioritní třídy procesu.
- Má alespoň jedno vlákno (primární - to hlavní), které může vytvářet další vlákna – threads a fibers

Thread

- Entita v rámci procesu, kterou **plánuje OS**. Všechny vlákna sdílí paměťový prostor a zdroje svého procesu. Vlastní handler výjimek. Priorita: OS zajišťuje inverze priorit, dynamic boost, foreground/input včetně realtime.
- Jedinečný identifikátor
- TLS – thread local storage, data, která jsou specifická/lokální pro daný thread. Možnost, jak si thread může zabezpečit jedinečný přístup ke svým datům. Každý proces má k dispozici několik TLS slotů, které mohou být použity jeho thready. Možnost využití ke zpracování výjimek.

Fiber

- Běží v kontextu vlákna, **plánuje ho thread procesu**, jeden thread může naplánovat několik fibers. FLS – fiber local storage =analogie k TLS, FLS je asociováno s threadem.
- Má malý kontext (v porovnání s kontextem threadu) – zásobník, podmnožinu registrů a inicializační data.
- Má přístup do TLS threadu, v jehož kontextu běží.
- Nemá prioritu.

Synchonizační objekty WinApi

Mutex – vyžaduje přepnutí do režimu jádra, CreateMutex, OpenMutex, ReleaseMutex

Event – stavy signaled/nonsignaled, může být buď pulsní, tj, překlopí se do nonsignaled jakmile propustí jednoho čekajícího, nebo zůstane signaled (Manual reset Event, Auto Reset Event). CreateEvent, SetEvent, ResetEvent.

Pozn. txkoutny říkal, že event je rychlejší kvůli tomu, že nevyžaduje přepnutí do režimu jádra - lhal. Mutex, Event i Semaphore vyžadují přepnutí, protože jsou implementovány na úrovni jádra a umožňují synchronizaci napříč různými procesy (mohou být pojmenované). Opět v přednáškách je jaksi uvedeno, že kritická sekce má velkou režii - není to pravda, kritická sekce jako jediná neumožňuje synchronizaci mezi procesy, ale pouze v rámci jednoho procesu. Díky tomu při běžném vstupu/výstupu z/do ní (pokud je volná) není nutné nikam přepínat. Do režimu jádra se přepne pouze v případě, že je kritická sekce obsazená a je nutné čekat na uvolnění - pak se přepne do režimu jádra a čeká se nad nějakým synchronizačním primitivem, např. EVENT nebo MUTEX.

http://www.codeproject.com/Articles/7953/Thread-Synchronization-for-Beginners?fid=89682&select=3191727&fr=31#xx0xx

For Windows, critical sections are lighter-weight than mutexes.
Mutexes can be shared between processes, but always result in a system call to the kernel which has some overhead.
Critical sections can only be used within one process, but have the advantage that they only switch to kernel mode in the case of contention - Uncontended acquires, which should be the common case, are incredibly fast. In the case of contention, they enter the kernel to wait on some synchronization primitive (like an event or semaphore).
I wrote a quick sample app that compares the time between the two of them. On my system for 1,000,000 uncontended acquires and releases, a mutex takes over one second. A critical section takes ~50 ms for 1,000,000 acquires.
Here's the test code, I ran this and got similar results if mutex is first or second, so we aren't seeing any other effects.

From <http://stackoverflow.com/questions/800383/what-is-the-difference-between-mutex-and-critical-section>

Semafor – klasický semafor, CreateSemaphore, OpenSemaphore, ReleaseSemaphore

Kritická sekce – Critical section, má velkou režii, EnterCriticalSection (blokující), TryEnterCriticalSection (neblokující), LeaveCriticalSection

Na signalizaci objektů lze čekat pomocí WaitForSingleObject nebo WaitForMultipleObjects.

Knihovny pro tvorbu paralelních aplikací bez nutnosti tvoření vláken v kódu:

OpenMP

Idea je taková, že vezmu sériový program a jen s minimem změn a s pomocí direktiv preprocesoru určím, které úseky kódu se budou vykonávat paralelně. Pokud se překladač vyvolá s direktivou OpenMP přeloží se jako paralelní program, jinak jako sériový. Pomocí direktiv pak lze překladači říci, které proměnné jsou sdílené a vyžadují tedy synchronizaci (kritickou sekci) a které nikoli.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Intel Thread Building Blocks.

Sunday, June 9, 2013 3:21 PM

TBB knihovna jako taková využívá vláken OS, ale programátor s nimi již nepracuje. Místo toho jsou v TBB tzv. Tasky – knihovna pak vykonává jednotlivé úlohy paralelně. Lze si napsat vlastní plánovač tasků a protože TBB nespoléhá jen na direktivy překladače, lze paralelizovat i takové úlohy, které by se s OpenMP paralelizovaly těžko.

Viz příloha z přednášek.

http://www.slideshare.net/psteinb/abhishek-sync-locks-gdc

```
  localMaxAbsDiff = max(localMaxAbsDiff,
                        sublocalMaxAbsDiff);
} //pragma omp parallel


if (cnt) {
  //if to avoid division by zero
  floattype tmp;
  tmp = valuescnt;
  tmp = 1.0/tmp;

  (*stats).AvgAbsDiff = localAvgAbsDiff*tmp;
  (*stats).AvgDiff = localAvgDiff*tmp;
  (*stats).MaxAbsDiff = localMaxAbsDiff;
}
```

Intel Thread Building Blocks

- Open Source, podpora více platforem
  - Stejně jako OpenMP
- Ačkoliv si je TBB vědoma vláken poskytovaných OS, myšlenka je takový, že programátor už s vlákny nepracuje
- Namísto vláken specifikuje úlohy, tasks, a jejich návaznosti

- Knihovna vykonává úlohy paralelně, jak nejlépe to jde
  - Obsahuje různé optimalizace v jakém pořadí úlohy vykonávat
  - Ne vždy platí, že FIFO je nejlepší strategie
    - Využívá se technika „task stealing"
  - Lze si napsat vlastní plánovač

- Stejně jako STL, i TBB intenzivně používá C++ šablony
  - Tj. nelze ji použít v C jako OpenMP

- Základní konstrukce TBB jsou
  - parallel_for, parallel_reduce, parallel_scan
  - parallel_while, parallel_do,pipeline, parallel_sort
  - paralelní kontainery, které jsou v STL
  - mutex, spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, recursive mutex
  - fetch_and_add, fetch_and_increment, fetch_and_decrement, compare_and_swap, fetch_and_store

- TBB dokáže použít vlastní paměťový manažer, který je optimalizovaný na paradigma výpočtu úloh

- TBB je náročnější se naučit, ale zase se to vyplatí na výkonu, pokud se začíná psát nová aplikace, než např. paralelizovat s OpenMP
  - Navíc TBB se nespoléhá na direktivy pro překladač, takže podmínka if OpenMP se dá realizovat libovolně složitá, nebo se dají udělat konstrukce, který by šli s OpenMP udělat jen velmi obtížně – např. cancellation výpočtu v OpenMP není

- Např. chceme-li spustit na pozadí několik výpočetně náročných úloh paralelně

```
tbb::task* CMasterCalculationTBBLogic::execute() {
 tbb::task_list list;

 for (int i=gaiFirst; i<=gaiLast; i++)
  list.push_back(*new(allocate_child()) CTask(i));

 set_ref_count(gaiLast-gaiFirst+2); //počet úloh +1
 spawn_and_wait_for_all(list);

 return NULL; //non-NULL by byla úloha, která by měla
}         //být spuštěna jako další/závislá na téhle
```

- Když bychom např. násobili vektor

```
class CMulVect {
    floattype *mA, *mB;
    int mLen;
public:
    floattype mProduct;

    CMulVect(floattype *a, floattype *b, int len) :
            mA(a), mB(b), mLen(len), mProduct(0.0) {}
    CMulVect(CMulVect& x, tbb::split) : mA(x.mA),
             mB(x.mB), mLen(x.mLen), mProduct(0.0) {}

    void join(const CMulVect& y) {
        mProduct += y.mProduct;
    }

    void operator()( const
                    tbb::blocked_range<size_t>& r ) {
        int r_end = r.end();

        //Taken from Intel's tutorial on TBB
        //by declaring these variables, we make it
         //obvious to the compiler that they
        //can be held in registers instead in memory
         // => speedup

        floattype *a = mA;
        floattype *b = mB;
        floattype sum = 0.0;

        for (int i=r.begin(); i!=r_end; ++i)
            sum += a[i]*b[i];

        mProduct += sum;
    }
};
```

```
floattype MulVect(floattype *a, floattype *b,
                  int len) {

    floattype result = 0.0;

    //Jsou data tak velká, aby se je vůbec
    //vyplatilo počítat paralelně?
    if (len<=rmSerialThreshold) {
       for (int i=0; i<len; i++)
          result += a[i]*b[i];
    } else {
        CMulVect muller(a, b, len);

        tbb::parallel_reduce(
          tbb::blocked_range<size_t>(0,len), muller);
        result = muller.mProduct;
    }
    return result;
}
```

Případová studie použití TBB

- Uvažujme výpočetně náročnou aplikaci nad rozsáhlými daty s GUI
- Jedno vlákno bude obsluhovat GUI
- Jedno vlákno bude obsluhovat I/O
  - Naivní přístup je jedno vlákno pro zápis a jedno pro čtení
  - Lepší je jenom jedno vlákno a použití asynchronních I/O operací
    - OS si je uspořádá sám pro lepší výkon
    - Např. disky mají Native Command Queuing
  - Ale co s výpočetní částí?
    - Určitě v tom bude alespoň jedno vlákno, aby výpočet běžel na pozadí a GUI bylo responsive

- Přístup s psaním vláken by znamenal postarat se
  - Vytváření a rušení vláken, což je další režie pro OS
  - Jejich správnou synchronizaci, což je náročné, jakmile se program stává složitější
  - Výkonnostně je rozdíl, jestli se synchronizuje v kernel-mode, nebo v user-mode address space
  - Výkonnostně hraje roli, zda se k datům přistupuje ze stejného procesoru – thread affinity

  - Optimální počet vláken odpovídá počtu procesorových jader v systému, což ale není přístup, který je vždy možný při psaní vláken
    - Např. počet vláken může odpovídat tomu, jaká je metoda výpočtu – problém škálovatelnosti
      - S TBB se taková vlákna nahradí úlohami

- S TBB
  - O výše uvedené problémy se TBB postará
  - Programátor napíše jedno vlákno, ve kterém pak spustí výpočet úloh TBB
  - Nicméně programátor si stále musí být vědom toho, jak se píší paralelní programy s vlákny, aby mohl správně používat synchronizační primitiva TBB
    - Optimálně se naprogramují pouze úlohy s tím, že každá úloha po dokončení vrátí další úlohu, která se má vykonat jako navazující
    - Jako bariéra na dokončení více úloh se pak použije task list
    - Úlohy však mohou sdílet proměnné, a proto je třeba znát teorii o psaní vláken
      - Aby byla představa, že 2 úlohy mohou, ale i nemusí, běžet paralelně, např. když se má správně použít mutex, podmínka...

Rendez-Vous, vč. select v Adě, a jeho porovnání s monitorem.

Thursday, May 30, 2013 8:33 AM

- Ada je objektově orientovaný jazyk se silnou verifikací typů (nelze implicitně přetypovat datové typy – např. void pointer)
- nepoužívá interpretr
- nepodporuje fibres
- paralelní části výpočtu se označují jako tasky (ne thready)
- mohou být prováděny na jednom procesoru, více procesorech nebo více počítačích, není to však vyjádřeno v kódu programu
- pro synchronizaci tasků se používá asymetrické synchronní rendezvous (zasílání zpráv)

Tasky

Konstrukce *task* představuje program paralelně proveditelného procesu, schopného komunikace s ostatními procesy.

Deklarace je následující:

```
task jméno is
deklarace jmen komunikačních typů
end jméno;
task body jméno is
lokální deklarace a příkazy
end jméno;
```

- pro ukončení procesu lze použít příkaz *abort jméno;*, ale jeho použití by mělo být výjimečné
- k ukončení tasku se používá příkaz *terminate*

```
Select
    Accept e
Or
    Terminate
End select
```

Rendez-vous

Pro interakci procesů používá ADA principu asymetrického rendezvous, kterým eliminuje potřebu semaforů (umí je nahradit), umožňuje synchronní a nepřímo (zavedením pomocného procesu) i asynchronní komunikaci procesů zasíláním zpráv. Dovoluje tak elegantní konstrukci monitorů.

- prostředek pro synchronizaci úkolů (tasks)
- dva úkoly spolu komunikují pomocí rendez-vous - Meeting point, entry calls
- task je uspán do té doby, než se dostaví druhý task, který s ním chce komunikovat

```
task Simple_Task is
entry Start(Num : in Integer);
entry Report(Num : out Integer);
end Simple_Task;
task body Simple_Task is
Local_Num : Integer;
begin
//čeká na vložení čísla – entry call
accept Start(Num : in Integer) do
Local_Num := Num;
end Start;
//normálně pokračuje v běhu
Local_Num := Local_Num * 2;
//čeká na vyzvednutí spočítané hodnoty
```

```
accept Report(Num : out Integer) do
Num := Local_Num;
end Report;
end Simple_Task;
```

- uvedený příklad stačí, pouze pokud potřebujeme jen jedno vlákno běžící podle uvedeného kódu
- průběh dostaveníčka - accept:
  - klient zavolá server
  - server si převezme parametry
  - server provede výpočet, klient spí
  - server předá výsledky

Select - může být nezbytné, aby úkol mohl reagovat na několik vstupních volání (entry calls) – pokaždé na jiné dle okolností – tj. ne v předem určeném pořadí

```
//Vynutíme si inicializaci a další se
//už pak může vykonávat v libovolném
//pořadí.
accept Init(Item : in Integer) do
    Local_Item := Item;
end Init;
loop
    select
        accept Stop;
                exit;
        or
        when podmínka = > //může i nemusí být
                accept Put(Item : in Integer) do
                    Local_Item := Item;
                end Put;
                Local_Item := Local_Item * 2;
        else
                Put_Line("No entry call at this time");
    end select;
    delay 0.01;
end loop;
```

Protected Objects, Protected Types

- tasky mohou sdílet objekty
- objekt je instance typu – klíčové slovo *type*
- klíčové slovo *protected* zajistí exkluzivní přístup k chráněnému objektu
- jsou tři operace nad chráněnými objekty:
  - **Procedury** – mění stav objektu, aniž by pro to musela být splněna podmínka; překladač se stará, aby měly exkluzivní přístup k objektu
  - **Entry calls** – stejné jako procedury, ale pro vykonání *entry call* je třeba navíc splnit podmínku
  - **Funkce** – pouze vrací stav a nic nemění, a proto nemusí mít exkluzivní přístup k objektu

Porovnání s monitorem

Rendezvous vs. monitors

The previous example is highly reminiscent of the monitor solution. Both provide “structured” approaches to mutual exclusion. Mutual exclusion is implicit in both monitors and rendezvous.

Passive and active objects

Monitors are **passive** objects.
• Only client threads exist.
• The client thread performs the service for itself inside the monitor.
• Server state does not change spontaneously.
Modules containing server threads are **active** objects.
• The server thread acts on behalf of the client thread within the module for the duration of the

rendezvous.
• Server threads may change the module state in between calls from clients.

http://www.engr.mun.ca/~theo/Courses/cp/pub/cp8-rendezvous.pdf - viz příloha

Renezvous v jave ze stranek zcu:
http://www.kiv.zcu.cz/research/groups/dss/download/presentation-2005-03-21.ppt

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx> no

Rendezvous

Rendezvous provides synchronization and two-way communication between two threads, a client and a server.

- From the client's point of view the rendezvous is a procedure call (remote or local)
- The client blocks until the server executes an in statement (MPDP) or accept statement (Ada).
- Server blocks until a call is made that it is prepared to accept.
- Once both are ready
  - arguments are copied to parameters
  - code is executed by the server
  - results are returned and "copy out" parameters are copied to arguments.
  - Both threads proceed independently.

Example

```
module TicketServer
        op getNext returns int ;
body
        process daemon {
                int next := 0 ;
                while( true) {
                        in getNext() returns val ->
                                val := next ;
                        ni
                        next := next + 1 ;} }
        end TicketServer
```

Note that mutual exclusion is implicit, since the server thread can be handling but 1 request at a time.

Choice

The server thread can offer a choice of requests that it will accept and can specify the conditions under which it will accept a choice.

```
module Bounded_buffer
        op deposit(char data);
        op fetch(result char data);
body

        process Buffer {
                char buf[n]; # buffer
                int front = 0; # first full slot
                int count = 0; # number of full slots

                while (true) {
                        in deposit(data) and count < n ->
                                buf[(front+count)%n] = data ;
                                count := count+1 ;
                        [] fetch(data) and count > 0 ->
                                data := buf[front] ;
                                front := (front+1)% n ;
                                count := count - 1 ;
                        ni } }
end Bounded_buffer
```

Rendezvous vs. monitors

The previous example is highly reminiscent of the monitor solution.

Both provide “structured” approaches to mutual exclusion.

Mutual exclusion is implicit in both monitors and rendezvous.

Passive and active objects

Monitors are passive objects.

- Only client threads exist.
- The client thread performs the service for itself inside the monitor.
- Server state does not change spontaneously.

Modules containing server threads are active objects.

- The server thread acts on behalf of the client thread within the module for the duration of the rendezvous.
- Server threads may change the module state in between calls from clients.

Data state vs. control state

With active objects, control state as well as data state can regulate operations that can proceed.

```
module Buffer
        op deposit( char data) ;
        op fetch( result char data) ;
body
        char buffer ;

        process daemon {
             while (true) {
                  in deposit(data) -> buffer := data ; ni
                  in fetch(data) -> data := buffer ; ni } }
end Buffer
```

- Acceptance of deposit and fetch strictly alternate.
- 2 states are represented by the program counter.
- Use of control state is sometimes clearer than use of data state.

Wait/signal vs. nested "in"

In monitors: a wait can happen at any point service is suspended until it can resume later.

With rendezvous: once a service has started it can only be suspended by a nested "in". Consider:

```
module Barrier {
        op done ;
body

        process daemon {
            while (true) {
                in done() -> in done() -> skip ni
                ni ; } }
end Sync2
```

How would you write an N process barrier?

Rendezvous vs. (Remote) Procedure Call

Rendezvous provides synchronization and mutual exclusion.

- RPC is handled by a new thread. Mutual exclusion must be made explicit.
- Rendezvous is handled by a single thread. Hence implicit mutual exclusion.

Rendezvous vs. Synchronous Message Passing

As with synchronous message passing the client (sender) is delayed until the server (receiver) is ready to accept the communication.

In a degenerate form

```
in MessType( param ) -> local := param ni
```

rendezvous is a synchronous receive. We abbreviate the above by

```
receive MessType( local ) ;
```

Rendezvous adds the ability for the server to

- delay the client until further processing is done and
- to send information back to client after that processing.

Rendezvous in Ada

The rendezvous is strongly associated with Ada since

- Ada introduced the concept (early ’80s)
- No other major language has supported it.

In Ada

- in is called accept.
- Operations are called entries.
- Choice requires use of select statement.

Conditional acceptance can not depend on parameter values.

An Ada Example

```
loop
        select when count < n =>
                accept deposit( data : in char ) do
                        buf((front+count) mod n) := data ;
                end deposit ;
                count := count+1 ;
        or when count > 0 =>
                accept fetch( data : out char ) do
                        data := buf(front) ;
                end fetch ;
                front := (front+1) mod n ;
                count := count - 1 ;
        end select ;
end loop ;
```

Non-blocking servers and clients (Ada)

The server thread can opt not to block.

```
loop
        select
                accept calibration( v : real ) do
                        scale := v ;
                end calibration
        else
                null ; - - do nothing
        end select
        sensorOut := scale * sensorIn ;
end loop
```

Likewise, the client can make a conditional call depending on whether the server is currently prepared to accept it.

```
select
        Queue.deposit( packet ) ;
else droppedPacketCount := droppedPacketCount + 1 ;
end select
```

Time outs

The server thread may time-out if it blocks too long.
A watch-dog thread

```
loop
        select
                accept AllIsWell ;
        or delay 10.0 ;
                RaiseAlarm ;
        end select
end loop
```

Likewise, the client may time out if service is not sufficiently fast

```
select Queue.deposit( packet ) ;
or delay 20.0 ;
        droppedPacketCount := droppedPacketCount + 1 ;
end select
```

Výpočetní prostředí s distribuovanou pamětí.

Thursday, May 30, 2013 8:33 AM

MPMD

Systém pro paralelní výpočet s distribuovanou pamětí se skládá z výpočetních uzlů a komunikačních kanálů.

- Univerzální počítačová síť (softwarový multipočítač, SETI)
- Univerzální paralelní počítač (stavěný přímo pro paralelní počítač, Cluster)
- Jednoúčelový paralelní počítač (jedna konkrétní aplikace s maximální optimalizací)

protože neexistuje sdílená paměť, používá se pro komunikaci mezi procesy především zasílání zpráv

protože systémy s distribuovanou pamětí nemají žádný úzký profil ve formě sběrnice, přes kterou by procesory přistupovaly ke sdílené paměti, hodí se pro úlohy vyžadující tzv. masivní paralelismus (stovky až tisíce procesorů)

Obecně systém s distribuovanou pamětí umožňuje větší urychlení než systém se sdílenou pamětí díky paralelizaci komunikace

- Zatímco se data přenášejí kanálem, uzel může počítat
- Urychlení ovšem závisí na dalších parametrech
  - Objemu interakce
  - Celkovém objemu zpracovávaných dat
  - Konkrétní hw architektura
  - Jak dalece je použitý programový kód optimální pro danou architekturu

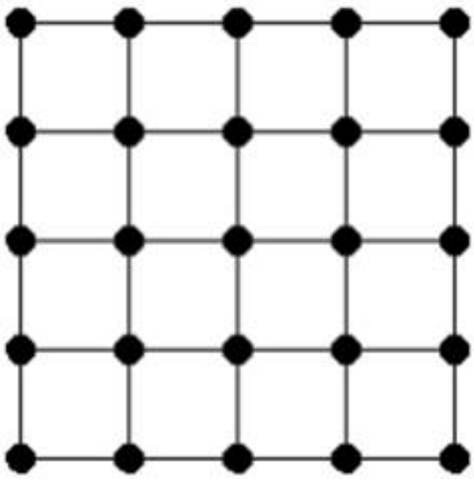
HW pohled

- Topologie obecně
  - **Pravidelná**: krychle, mřížka, hvězda
  - **Nepravidelná**: Internet

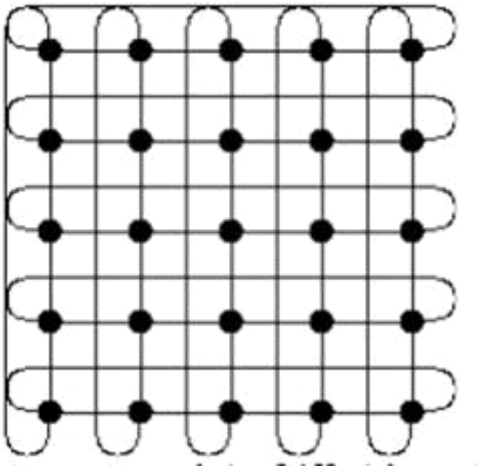
Směrování
Pevná topologie – můžete poslat vzkaz pouze sousedovi
Podle příjemce – směrování jak ho známe např. z IP
Podle odesílajícího – odesílající si určí cestu
  - IP „strict source and record route“ (SSRR)
  - IP „loose source and record route“ (LSRR)
    - Často blokováno z bezpečnostních důvodů

– address spoofing (podvrhnutí adresy)

- Fyzická topologie
  - Pevná: procesory jsou propojeny komunikačním kanálem
    - **Adresa může reflektovat polohu v síti**
    - Každý s každým
    - 2d mřížka (dmax = $2(n-1)$)



- Toroid (dmax = n-1)



- 3d mřížka = krychle
- n-rozměrné krychle…

  - Flexibilní: přepínání okruhů, přepínání paketů
    - Circuit switching
    - Packet switching

Parametry

- N: Celkový počet uzlů v síti
- dij: vzdálenost mezi dvěma uzly (sousedé mají 1)
- dmax: nejhorší varianta, kolika uzly musí projít zpráva, než je doručena (nejdelší cesta zprávy v celém systému)
- počet sousedů: s kolika dalšími uzly je daný uzel spojen přímo
- přenosová kapacita:
  - agregovaně – kolik uzlů může najednou posílat zprávu
  - odolnost proti chybám – kolik komunikačních kanálů musí selhat, než se z jedné sítě stanou dvě

Snahou je dosáhnout

- Co největšího počtu uzlů v síti
  - Škálovatelnost
- Co nejmenší komunikační vzdálenosti – dmax
  - Tj. omezit komunikační zpoždění
- Co nejmenší počet sousedů
  - Aneb, i komunikační kanál něco stojí
- Dosáhnout co největší přenosové rychlosti

- Cílem je namapovat virtuální topologii na síťovou tak, aby docházelo k co nejmenšímu zpoždění
  - Ideálně se fyzická, síťová i virtuální topologie shodují 1:1
  - „Úplně ideálně“ jde zároveň i o optimální metodu výpočtu (který používá danou topologii)
  - Např. mřížka na mřížku o stejných, nebo větších, rozměrech
  - Úlohy tzv. geometrické dekompozice v rovině
  - Model Farmer-worker/Master-slave
    - Farmer úkoluje workery
    - Virtuální topologie je typu hvězda

SW pohled

- **Alokování uzlů**
  - 1 proces na 1 uzel
    - Např. pevně daná u paralelního počítače
    - 1 proces dokáže plně využít celý uzel, takže nemá smysl jich na jednom uzlu spouštět několik
    - OS uzlu neumí spustit více jak jeden proces najednou
  - Potenciálně nula až několik procesů na jeden uzel
  - Přidělení celé sítě pro jeden výpočet
    - Celkový čas výpočtu je pak dán
      - Dobou k zavedení programů, spuštění procesů a distribuce dat do uzlů
      - Vlastním výpočetem
      - Získáním výsledků z uzlů
  - Přidělení části sítě jednomu výpočtu
  - Několik paralelně běžících výpočtů
    - Na jednom uzlu může běžet několik procesů
    - Nelze se spoléhat na odvozená urychlení, protože ta nepočítala se zátěží, kterou vygeneruje neznámý kód
    - Nehodí se pro synchronní/lockCstepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku
- **Identifikace procesů**
  - Jedinečná ID procesů
  - Interakce send/receive (vše ostatní je na nich postaveno)
  - Podle přidělení na uzly:
    - 1 uzel – 1 proces
    - Více procesů na uzlu
    - Více procesů na uzlu a procesy mohou migrovat (tabulka umístění procesů)
- **Komunikační schéma**
  - **Fyzická topologie** = *jak je to sdrátováno*
  - **Síťová topologie** = *jak vidí fyzickou topologii software*
  - **Virtuální topologie** = *komunikační vazby procesů*
  - Ideálně 1:1 (aby docházelo k nejmenším zpožděním)
- Virtuální topologie může mít:
  - Pravidelnou strukturu - mřížka, hvězda…
  - Nepravidelnou strukturu
- Může být
  - Statická - např. mřížka
  - Dynamická - procesy mohou procházet různými fázemi, mohou vznikat i zanikat

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Rozdíly mezi PVM a MPI.

Thursday, May 30, 2013 8:34 AM

- PVM a MPI se používají v prostředí s distribuovanou pamětí

Hlavní rozdíly

  - *PVM* - vznik v prostředí *heterogenní* sítě, *MPI* navržen spíše pro clustery - *homogenní* prostředí (u obou je ale běh možný v heteregonním prostředí)
  - MPI - jednodušší a efektivnější abstrakce na vyšší úrovni
  - MPI nabízí bohatší možnosti pro přenos zpráv (např. plně duplexní *sendrecv,* perzistentní komunikace, každý pošle každému - *MPI_Alltoall*)
  - MPI se už v návrhu snaží o omezení kopírování paměťových bloků
  - PVM se nestará o topologii, MPI podporuje logické komunikační topologie
  - MPI nemá žádný config jako PVM
  - MPI se vyhýbá nízkoúrovňovým rutinám kvůli přenositelnosti
  - MPI - možnost definovat vlastní datové typy pro snížení režie při posílání velkého množství dat (vs. PVM - vícenásobné volání *pvm_pack*)
  - MPI - podpora souborových operací (soubor se rozdělí na 1-N bloků, čtení a zápis jako zasílání zpráv)

PVM (Parallel Virtual Machine)

- Univerzální výpočetní model pro **heterogenní** distribuované výpočetní prostředí
- v PVM se musí provádět operace PVM_initsend, PVM_PK* apod.
- PVM umožňuje především provádět distribuované výpočty v heterogenním prostředí (různé architektury)
- PVM se nestará o topologii, MPI podporuje logické komunikační topologie.
- Programátor PVM může využít funkci PVM_Config aby např. určil, kde spustit další proces – MPI nic takového nemá.
- PVM programátorovi umožňuje, aby se do systému napojil pomocí nízko úrovňových rutin, MPI se tomu vyhýbá kvůli vyšší přenositelnosti.

Základní funkce

- Probíhá pomocí zasílání zpráv

pvm_spawn(char *task, char **argv, int flag,char *where, int ntask, int *tids)

- Spustí několik procesů podle zadaného programu
- Procesy se po vytvoření neznají -> proces, který je vytvořil je musí seznámit
  - numt – počet úspěšně spuštěných procesů (*návratová hodnota)*
  - task – spustitelný soubor
  - argv – parametry
  - flag – možnosti spuštění
    - PvmTaskDefault – PVM si rozhodne, kde spustit
    - PvmTaskHost – where bude obsahovat adresu, kde spustit
    - PvmTaskArch – where bude obsahovat typ architektury/platformy
    - Existují i další jako PvmTaskDebug, PvmTaskTrace, PvmMppFront, PvmHostCompl
  - where – kde spustit proces, ignoruje se, pokud nejsou příslušně nastaveny možnosti spuštění
  - ntask – počet procesů ke spuštění
  - tids – identifikátory spuštěných procesů

pvm_initsend(int encoding) – nastavení kódování (defaultně se používá PvmDataDefault)

int pvm_pkint(int *ip, int nitem, int stride) – vloží data do bufferu

int pvm_send(int tid, int msgtag) – pošle data procesu (TID)

int pvm_recv(int tid, int msgtag) – přijme data od procesu (TID)

pvm_upkint(int *ip, int nitem, int stride) – rozbalí data

MPI (Message Passing Interface)

- MPI je postaveno na PVM
- Knihovna pro podporu paralelních výpočtů v systémech s distribuovanou pamětí, vazby (interface) má pro programovací jazyky C a Fortran.

- Proti PVM se jedná o programovací prostředek vyšší úrovně, vztah je asi jako mezi jazykem symb. adres (odpovídá PVM) a vyšším programovacím jazykem (odpovídá MPI), tedy:
  - v PVM jde naprogramovat "skoro cokoliv", ale dá to velkou práci a program pak nejspíš nebude přenositelný,
    - dominantní aplikací pro MPI jsou numerické výpočty s regulárními daty (vektory či matice s číselnými prvky), přičemž pro takové aplikace je programování relativně pohodlné a program je dobře přenositelný do jiné instalace MPI

- MPI je primárně míněno (na rozdíl od PVM) pro **homogenní** výpočetní prostředí (tj. cluster stanic se společnou administrací, superpočítač jako N-Cube, ap.), takže poskytuje prostředky i pro "synchronní" algoritmy (tj. takové, kde se předpokládá přibližně stejná rychlost běhu jednotlivých procesů) a odpovídající statické rozdělení práce mezi procesy.

- MPI primárně využívá SPMD model paralelního výpočtu, tj. vyrobí se (na rozdíl od PVM) jen jeden "exe" soubor programu a ten se zavede do zvoleného počtu (dále N) procesorů. V každém procesoru běží jen jeden proces. Všech N procesů tudíž běží podle téhož programu, zjistí si své číselné ID a podle toho odliší svou činnost. V zásadě je tudíž realizovatelný i MPMD model výpočtu (třeba ve verzi farmer-workers, přičemž v programu je přepínač podle ID, jednu větev realizuje proces s číslem třeba 0 - farmer, druhá větev (jiná čísla než 0) je pro procesy typu worker).

- Komunikace procesů je asynchronní message-passing s přímým adresováním přes číselné ID procesu. Existuje mechanismus skupin procesů (viz dále tzv. komunikátory) a možnost broadcastu zprávy ve skupině procesů. Zprávy není na rozdíl od PVM třeba pracně "pakovat". MPI má svoje "primitivní datové typy" a z nich lze skládat "strukturované typy", sloužící ovšem jen pro účely komunikace (tj. zjednodušený popis toho, co má přijít do zprávy - lze srovnat s náročností "pakování" zprávy v PVM).

- Existuje sada funkcí pro tzv. "globální operace", tj. operace nad daty, jejichž instance jsou "rozprostřeny" ve všech procesech výpočtu. Realizace takových operací v PVM se musí pracně rozepsat do primitivnějších operací pvm_send() a pvm_recv().

Základní funkce

Je jich 6, přičemž už umožňují napsat jednodušší aplikaci. První čtyři z nich je třeba použít v každém MPI programu. Všechny funkce vrací celočíselný kód úspěšnosti provedené operace, přičemž symbolická hodnota MPI_SUCCESS znamená úspěch. Přehled základních funkcí v C-syntaxi:

int MPI_Init (int* argc, char* argv)**

- Inicializace MPI výpočtu, argc a argv jsou argumenty hlavního programu.

int MPI_Comm_size (MPI_Comm comm, int* adr_size)

- Zjištění počtu procesů, počet se dosadí do proměnné odkazované parametrem adr_size. MPI_Comm je typ "komunikátor", pokud při volání dosadíme za parametr comm hodnotu MPI_COMM_WORLD,

zjišťujeme počet všech vytvořených procesů aplikace N.

int MPI_Comm_rank (MPI_Comm comm, int* adr_rank)

- Zjištění čísla procesu v rámci "komunikačního světa" comm. Čísla jsou v rozmezí 0 až M-1, kde M je "rozměr komunikačního světa" reprezentovaného komunikátorem comm (M = N, pokud dosadíme za comm hodnotu MPI_COMM_WORLD).

int MPI_Finalize (void)

- Ukončení výpočtu v MPI, provádí každý proces.

int MPI_Send (void* adr_buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

- Odeslání zprávy s typem tag v komunikačním světě comm procesu dest. Odesílá se zpráva z bufferu buf obsahující count položek typu datatype.
- Čili buffer pro zprávu je na rozdíl od PVM kdekoliv v datech programu (zadá se adresa příslušného pole). Za datatype se dosazují buď primitivní typy MPI, například MPI_INT, MPI_DOUBLE, MPI_CHAR, nebo strukturované typy vytvořené z primitivních (viz dále).

int MPI_Recv (void* adr_buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *adr_status)

- Blokující příjem zprávy. Parametry mají analogický význam jako u MPI_Send. Navíc je parametr adr_status, odkazující kam se má uložit status příjmu zprávy (výstupní parametr). Status obsahuje položky: status. MPI_SOURCE - od koho zpráva přišla a status.MPI_TAG - jakého typu zpráva přišla. Dosadí-li se za source hodnota MPI_ANY_SOURCE a za tag MPI_ANY_TAG, přijme se jakákoliv zpráva a z uvedených položek výstupního parametru status se dá zjistit co to vlastně přišlo.

Globální operace MPI

- Globální operace jsou operace, do kterých jsou zapojeny všechny procesy patřídí do téhož "komunikačního světa" (tj. jedním parametrem funkcí pro globální operace je příslušný komunikátor). Funkci globální operace volají všechny zúčastněné procesy (což je pochopitelné, protože typicky procesy běží podle téhož programu), přičemž jejich činnost se v obecném případě liší podle čísla procesu.
- Globální operace v PVM nejsou (kromě broadcastu) a musely by se pracně rozepsat do posloupnosti operací send() a receive().

Globální operace MPI lze rozdělit na tři skupiny - synchronizace, přesuny dat a redukční operace.

Synchronizace

Každý komunikátor mj. realizuje bariéru, na které se mohou všechny jeho procesy synchronizovat voláním funkce

int MPI_Barrier (MPI_Comm comm)

  - Volání této funkce je tedy blokující a výpočet každého procesu pokračuje následujícím příkazem až poté, kdy se na bariéře "sejdou" všechny procesy patřící do "komunikačního světa" comm.

Přesuny dat

- Jedná se o operace s charakterem broadcastu, shromáždění či rozptýlení dat a tzv. redukční oprace.

int MPI_Bcast (void* adr_buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

- Operace broadcast. Má v zásadě stejné parametry jako MPI_Send(). Proces s číslem root vysílá, ostatní zprávu přijímají (čili root používá buffer jako vysílací, ostatní jako přijímací). Proti send() chybí tag - pro "globální" komunikační operaci nemá význam - všichni aktéři komunikace prochází tímtéž bodem programu a tudíž vědí, "o co při komunikaci jde".

int MPI_Gather (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)

- Proces s číslem root shromažďuje data od všech procesů včetně sebe. Čili všechny procesy vysílají data (count položek typu intype) z bufferu inbuf a proces root je zapisuje do bufferu outbuf - samozřejmě je zapisuje v pořadí podle čísel vysílajících procesů (tj. nikoliv tak, jak zprávy došly). Parametr outbuf (a též outcnt a outtype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

int MPI_Scatter (void* adr_inbuf, int incnt, MPI_Datatype intype, void* adr_outbuf, int outcnt, MPI_Datatype outtype, int root, MPI_Comm comm)

- Inverzní operace k Gather(), tj. proces s číslem root "rozptyluje" data z bufferu inbuf všem ostatním procesům včetně sebe. Vstupní buffer musí obsahovat M částí dat (obecně různých, jedna část je pole count položek typu intype), přičemž i-tá část se pošle do bufferu outbuf procesu s číslem i. Parametr inbuf (a též incnt a intype) využije jen proces root. Za incnt a intype se normálně dosadí stejné hodnoty jako za outcnt a outtype.

Redukční operace

Redukční operace má M operandů umístěných ve vstupních bufferech komunikujících procesů. Výsledek operace se zapisuje do výstupního bufferu buď jednoho určeného procesu (root) nebo všech procesů.

int MPI_Reduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Za parametr op se dosazuje typ realizované operace, kde použitelné symbolické hodnoty typu jsou

- MPI_MAX, MPI_MIN (maximum, minimum),
- MPI_SUM, MPI_PROD (součet, součin),
- MPI_LAND, MPI_LOR, MPI_LXOR (logické operace)
- MPI_BAND, MPI_BOR, MPI_BXOR (logické operace se všemi bity)

Operandy jsou ve vstupních bufferech procesů, výsledek je ve výstupním bufferu procesu root.

int MPI_Allreduce (void* adr_inbuf, void* adr_outbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

- Funguje jako předchozí operace, ale výsledek dostanou do výstupního bufferu všechny zúčastněné procesy, tudíž chybí parametr root, který pak nemá smysl.

Komunikátor

Komunikátor je interní objekt MPI (typ MPI_comm, jedná se o typ tzv. "handle", čili jakéhosi zobecněného ukazatele reprezentujícího komunikátor).

Komunikátor reprezentuje skupinu komunikujících procesů a komunikační kontext této skupiny.

Dále komunikátor slouží pro paralelní členění programu, či jinak řečeno pro realizaci modelu MPMD (funkční paralelismus), kdy se procesy aplikace rozdělí na skupiny (reprezentované různými komunikátory) a každá skupina "dělá něco jiného" a její procesy "se vybavují jen mezi sebou". Čili uvnitř skupiny procesů (komunikátor) funguje datový paralelismus, kdežto mezi skupinami procesů funguje funkční paralelismus.

Základní funkce nad komunikátory jsou (podrobnosti viz literatura):

MPI_Comm_rank()

- vrací počet procesů komunikátoru, základní funkce MPI - viz dříve v části 2

MPI_Comm_dup()

- vytváří duplikát komunikátoru

MPI_Comm_split()

- "štěpí" kpomunikátor na dva jiné, čili rozděluje jednu skupinu procesů na dvě jiné

  MPI_Comm_free()

- uvolňuje (likviduje) komunikátor (samozřejmě nikoliv procesy, které komunikátor reprezentuje)

  MPI_Intercomm_create()

  vytváří tzv. "interkomunikátor" jakožto prostředek komunikace mezi dvěma skupinami procesů.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Přidělování práce v prostředí s distribuovanou pamětí, možnosti urychlení výpočtu a přiřazení procesů na jednotlivé uzly.

Thursday, May 30, 2013 8:34 AM

Přidělování práce v prostředí s distribuovanou pamětí

Pozn. Přednáška z PPR b_advance.

Faktory ovlivňující rychlost výpočtu

- Virtuální topologie, komunikační schéma, distribuované aplikace
- Prezentovaná síťová topologie
- Fyzická topologie sítě
- Výkonnost jednotlivých uzlů v síti
- Výpočetní model distribuované aplikace
    - Každý proces může běžet podle vlastního programu

Řešení obecně

- Umístit procesy na uzly sítě takovým způsobem, aby běžely co nejrychleji a zároveň bylo komunikační zpoždění co nejmenší
    - Může se jednat i o protichůdné požadavky
- Zatížení a dostupnost uzlů se může měnit v čase
- Jsou tři typy úloh
    - S konečným časem výpočtu – distribuovaná aplikace má jenom něco spočítat a pak skončí
    - S teoreticky nekonečným časem výpočtu – distribuovaná aplikace poskytuje službu
        - Neplatí, že uzel musí být zcela vytížen výpočtem po celou dobu
    - Processing While in Transit – výpočet se nad daty provádí během jejich přenosu Např. Active Network

Možnosti urychlení

MPMD

multiple processes, multiple data

- Ve všech uzlech není stejný program
- Každý proces má svoje data

SPMD

- Do všech uzlů se zavede stejný program
- Do uzlů se zavedou specifická data procesů
- Každý proces dokáže zjistit svoje ID a z něj určí sousedy a svůj díl dat
- Jeden z procesů je řídící
    - Obvykle ten první vytvořený
        - ⇨ Mívá ID 0, ale konkrétně to závisí na sw
    - Zpracovává dílčí mezivýsledky
- V homogenním distribuovaném prostředí se zjistí celkový objem dat, počet spuštěných procesů a práce se přidělí najednou
    - Např. cluster z identických stanic a MPI
- V heterogenním prostředí je lepší využít dynamické přidělování práce
    - Uzly nemusejí být stejně výkonné
    - Ale i v případě, kdy uzly nejsou využívány jednouživatelsky
    - Např. model farmer-workers, kdy farmáři udělíme čestný titul identický program -> bude k tomu všemu ještě makat jako worker

Urychlení u Farmer-Worker v distribuovaném prostředí, pokud farmer jen kompletuje dílo od

workerů a pokud zanedbáme odeslání a příjem zprávy je přibližně S ~ N - 1, tj. přibližně lineární

Hodí se tehdy, když:
- Datový objem zpráv je malý
- Výpočet je v porovnání s objemem zpráv náročný
- Např. nemá smysl počítat pole integerů k součtu na jiný uzel, v dnešní době je **operace mov zhruba stejně časově náročná jako add, muselo by se tedy vykonat spoustu operací navíc** => zpomalení

V heterogenním prostředí se realizuje automatický **load balancing** viz dále, urychlení pak závisí na velikosti dat ke zpracování jedním procesem, je třeba najít ideální objem dat, který příliš nezpomaluje a zároveň poskytuje dobré urychlení
- Příliš malé objemy dat nevedou k urychlení
- Příliš velké objemy dat k urychlení už vedou, ale zase se zbytečně příliš čeká na procesy, které z různých důvodů pracují pomaleji než ostatní

Ideální velikost posílaných dat tak závisí na:
- Výpočetním výkonu uzlů
- Měla by uzel zaměstnat na tak dlouho, aby bylo možné úkolovat uzly v době, kdy ostatní počítají
  - V první vlně přidělit náhodné velikosti dat od jednoho až dvou násobků objemu zprávy
    - ⇨ Tím se na nějakou dobu zabrání konvergenci, kdy bude komunikační linka využívána všemi procesy
      - □ Eliminace komunikačního zpoždění překrytím výpočtem
        - ◆ Kromě neblokujících komunikačních operací
- Data by se měla spočítat v nějakém přiměřeném čase, aby se na poslední proces nečekalo celou věčnost

MPSD
- Step-locked
- Pásová výroba (pipeline)
- Části dat by měly být stejně velké ne podle objemu dat, ale výpočetně
- Problémem může být kapacita přenosových kanálů
  - Objem dat může být příliš velký a tak uzel může nějakou dobu čekat na data
  - Ideálně se v i-tém kroku
    - Počítají data
    - Data z i-1 kroku se posílají dalšímu uzlu v řadě
    - Přijímají se data, která se budou počítat v i+1 kroku
    - Překrytí doby komunikace dobou výpočtu => eliminace komunikačního zpoždění, pokud se data déle počítají, než přijímají
      - ⇨ Pokud ne, zmenšit objem posílaných dat
- N - počet úkolů
- Pošleme-li objem vstupních dat k nekonečnu, pak je urychlení N

Load-Sharing
- Předpokládá se prostředí pracovních stanic, které nemusejí být vždy plně vytíženy
- Jeden uzel se vyhradí jako master, kde se spustí aplikace
  - Ostatní uzly se označují termínem slave
- V okamžiku, kdy je master vytížen na maximum, zkusí se vyhledat nevytížený uzel

Červ

- jednotlivé procesy se mohou replikovat na nevytížených uzlech
- červ se skládá z několika segmentů – procesů
- počet segmentů je buď pevně stanoven, nebo se při pokročilejší implementaci stanovuje dynamicky podle okolností
- nápadně připomíná šíření virů
- červ musí být věrohodný pro uživatele pracovních stanic

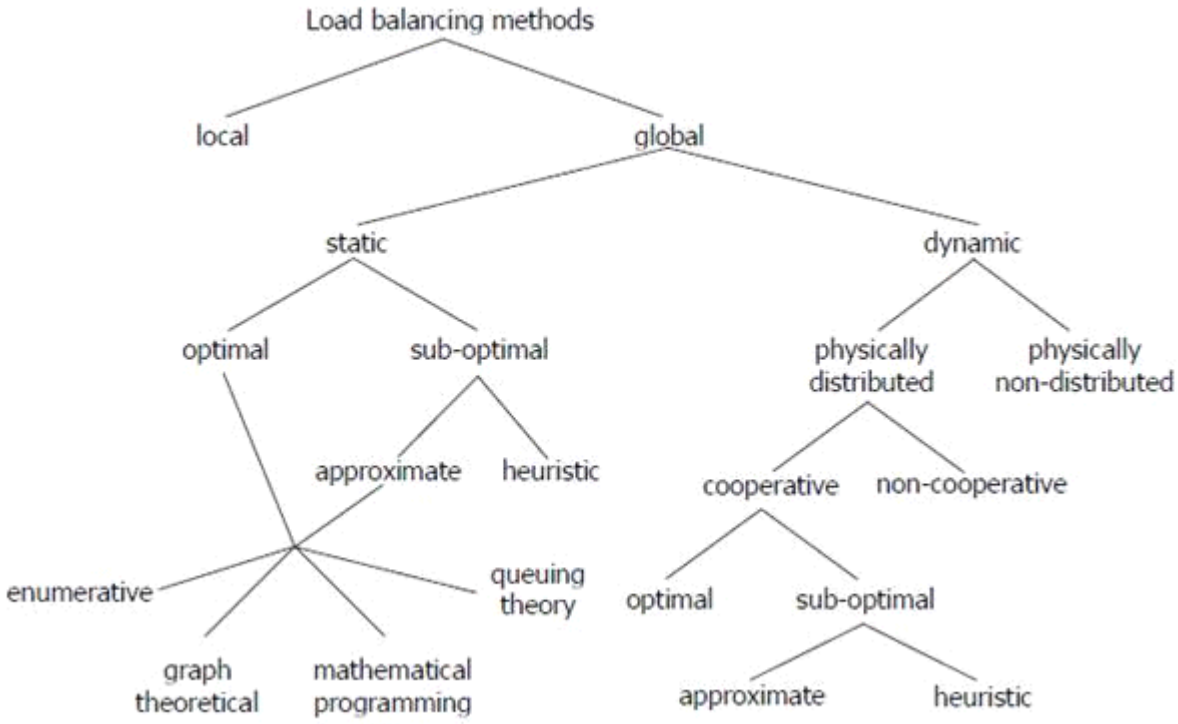
- komponenty červa
  - inicializační kód ke spuštění master
  - inicializační kód ke spuštění slave
  - výpočetní program
  - ze života červa
    - nalezení nevytíženého uzlu
  - žádný uzel nezná globální stav sítě, a proto se každý segment musí postarat o nalezení volné stanice sám za sebe
  - lze hledat například pomocí broadcast hodila by se synchronizace, protože několik segmentů může soupeřit o stejný uzel
    - uvolnění uzlu
  - segment se musí postarat, aby uzel byl zase viditelný jako dostupný i pro ostatní
    - kontrola růstu
  - čím větší červ, tím vyšší rychlost výpočtu, ale vznikají další problémy
- rychlost nemusí růst lineárně
- synchronizace
- stabilita

Condor

- snaží se o fér využívání všech uzlů
- pokud některý uzel selže, proces se restartuje jinde
- arbitr, který rozhoduje o tom, kde se spustí proces
  - sám hledá použitelné uzly
  - centralizované místo
- možnost selhání
- checkpoints
  - procesy lze relokovat za běhu distribuované aplikace
- používá se ukládání obrazu procesu v paměti,
- ne rekonstrukce stavu
- uzly musejí být identické
- co s otevřenými soubory?
  - checkpoint se ukládá periodicky
- pokud selže výpočet, použije se poslední
- checkpoint
  - pre-emptivnost procesů je implementována pomocí checkpointů

Load-Balancing

- na rozdíl od load-sharingu se předpokládá, že celá síť je dostupná pro výpočet
- existuje několik různých metod, které se liší použitelností, účinností, náročností na zdroje (paměť, procesor, …), přesností, spolehlivostí, …



- statické
  - výpočet přiřazení procesů na uzly je proveden ještě před spuštěním distribuované aplikace
    - výpočet může běžet libovolně dlouho, abychom dosáhli požadované přesnosti předpovědi – pokud ji metoda umožňuje dosáhnout
  - nelze reagovat na dynamické změny v prostředí
  - vyžadují předem spoustu informací o chování sítě a aplikace (např. kom. zpoždění, doby běhu procesů)
    - nereálné požadavky nelze splnit
      - ⇨ vliv na přesnost a tedy i rychlost výpočtu
- dynamické
  - výpočet přiřazení procesů na uzly sítě se provádí za běhu distribuované aplikace
  - výpočet se odehrává v reálném čase a nemůže si proto dovolit konzumovat příliš mnoho zdrojů
    - umí se vyrovnat s dynamickými změnami
    - ⇨ procesy musí umět pre-empci
    - ⇨ potřebné informace lze zjistit až za běhu aplikace, nebo si jich část vyžádat předem
- pre-emptivní
  - procesy lze přerušit během výpočtu a přemístit je na jiný uzel, aby bylo možné kompenzovat změny v síti
  - např. některý z procesů mohl skončit svoji činnost, nebo se odebral do dlouhodobého wait-stavu
  - pokud tuto vlastnost procesy nemají, na změny lze reagovat až při vytváření
- centralizované
  - mají jeden centrální prvek, arbitr, který rozhoduje o rozdělování zátěže na jednotlivé uzly
  - centrální prvek je slabé místo, co se stane, když selže?
    - ⇨ centralizované správa vyžaduje komunikaci jednoho uzlu se všemi
  - možnost přetížení
  - arbitr má přehled o známé síti, a proto lze očekávat, že dokáže zátěž rozdělovat celkem efektivně bez rizik, která jsou jinak spojena s distribuovaným principem
- distribuované
  - rozhodování o rozdělování zátěže provádí několik až všechny procesy
  - mohou být distribuovány na několik uzlů
  - když jeden selže, nic se neděje, pokud ho výpočetní model aplikace nutně nepotřebuje k životu
  - procesy, které provádějí rozhodování mohou být buď specialisté, anebo to mají jako

„vedlejšák“ ke své hlavní činnosti

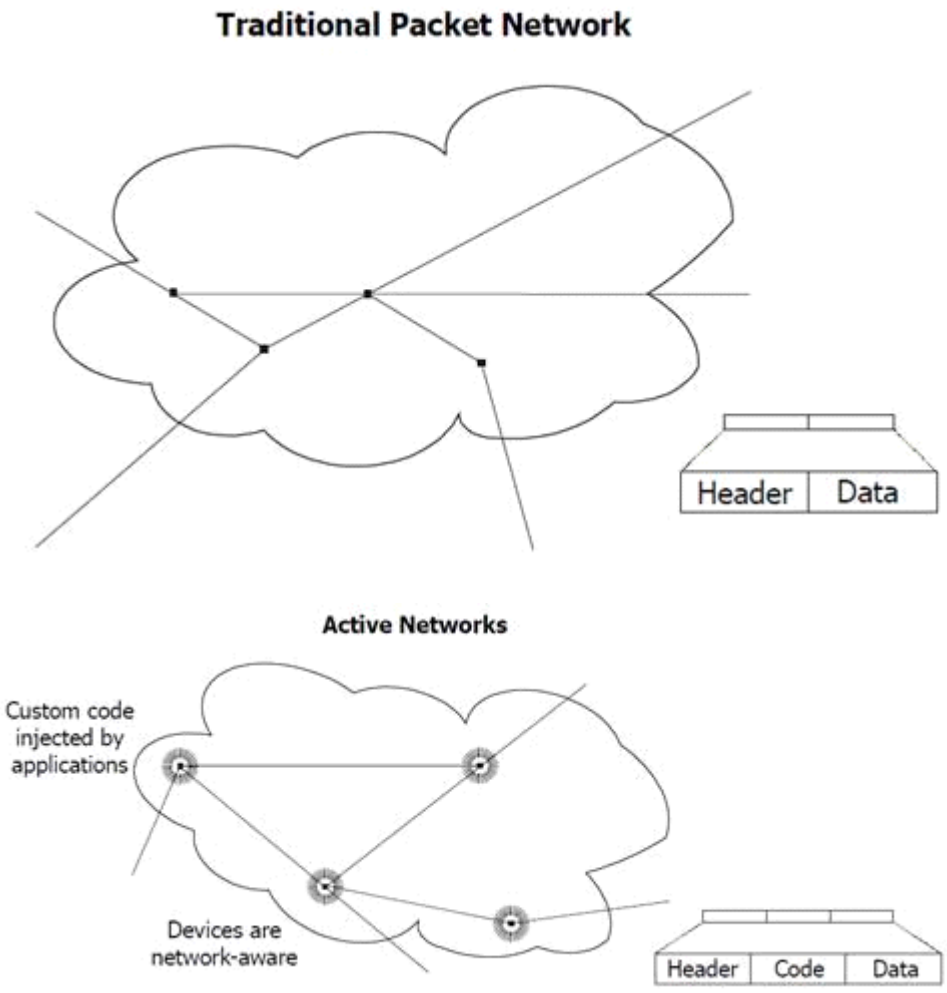
- adaptivní
  - síť prochází změnami během výpočtu – mění se stav uzlů
  - adaptivní metody berou do úvahy i několik předchozích stavů při rozhodování o přidělení zátěže
- kooperativní
  - každý proces se může rozhodovat buď sám za sebe, nebo může na rozhodnutí spolupracovat s ostatními
  - přímo – procesy spolupracují nad konkrétním rozhodnutí
  - nepřímo – procesy dávají informace o svých rozhodnutích k dispozici ostatním a ty je použijí při svých rozhodnutích
- sender-initiated
  - v okamžiku, kdy je uzel zatížen přes určitou mez, začne vyhledávat jiné uzly, kam by přemístil část své zátěže
  - je to režie navíc, protože čas potřebný na vyhledávání nových uzlů mohl být použit na běh procesů
- receiver-initiated
  - v okamžiku, kdy zátěž uzlu klesne pod určitou mez, začne vyhledávat jiné uzly, odkud by mohl převzít jejich zátěž
  - režie se projeví zvýšenou komunikací, uzel má dost volného výpočetního času, který může alokovat pro vyhledávání zátěže

Load Redistribution

- load-balancing tradičně měří zátěž v počtu procesů, což není zrovna to nejlepší
- následující text bude o Load-Redistribution Method in Distributed Environment
- metoda vyžaduje pokročilou síťovou architekturu jako jsou Aktivní sítě

Aktivní sítě

- stvořil Pentagon pro vyřešení nedostatků IP protokolu
- např. *Any-Cast* = metodologie pro adresování a routování, kdy jsou datagramy od jediného odesilatele routovány uzlu, který je topologicky nejblíže v dané skupině potenciálních příjemců (ačkoliv může být zasláno vícero uzlům, pokud mají stejnou adresu). Rozdíl od multicastu, který posílá všem ze skupiny najednou (a vždy), broadcastu, který zasílá jednoduše všem.
- Any-Cast je až v IPv6, aktivní sítě mají PAMcast – Programmable Any-Multicast – služba pro doručování zpráv, která generalizuje jak anycast tak multicast , která doručuje zprávy M z N příjemců, kde 1 <= M <= N
- IP
  - Dvojice vysílající a příjemce
  - Vysílající odešle paket na konkrétní adresu, včetně portu, kde předpokládá příjemce
  - Pokud tam není, paket se zahodí a vysílající zjistí chybu až timeoutem
- Aktivní síť
  - Paket, nazývaný capsule, je asociován s kódem, který se spustí na každém uzlu, kterým capsule prochází
    - Vždy je přítomný příjemce
  - Kód může manipulovat s daty, vlastnostmi (cíl, TTL, atd.) a vykonávat další uživatelsky definované činnosti
  - Proces se označuje termínem aktivní aplikace
    - Distribuovaná aktivní aplikace se skládá z několika aktivních aplikací, které mohou injektovat capsule a zároveň capsule může injektovat aktivní aplikace



- Komunikační model sám-sobě
  - Je možné injektovat capsuli do sítě, aby nasbírala potřebná data a pak je předala procesu, který ji injektoval
  - U IP by bylo nutné mít dopředu na každém uzlu, který by capsule mohla navštívit, spuštěný specializovaný proces
- Migrace procesů
  - Migrující proces změní svoji síťovou adresu, ale ještě ji nedal na vědomí ostatním procesům
  - Informaci o své nové síťové adrese zanechal na uzlu, odkud migroval
  - Capsule, která má doručit data, dorazí na uzel, odkud proces odmigroval, tam ho nenajde, ale použije svůj kód, aby si přečetla novou adresu a pouze změní svůj cíl
  - Ostatní procesy si mohou aktualizovat záznamy až později – lazy update, u IP je nutné vyřešit předem
- V aktivní síti je zapotřebí standardizace pouze dvou věcí
  - Programového kódu
    - Kód vykonává Execution Environment (EE), na jednom uzlu může být několik EE
  - Code distribution protocol
  - Vše ostatní je pak už aplikačně specifické
- Při přerozdělování zátěže (load redistribution) se procesy rozhodují samy za sebe, periodicky sledují své okolí (jako kolonie organismů)
  - Dosažení vyváženého stavu ne hned, ale postupně
- Capsule zjistí síťové okolí uzlu z hlediska topologie, výkonnosti, zatížení, komunikačního zpoždění apod.
  - Využije se při hledání výhodnějšího uzlu pro odmigrování
- **Rizika:**
  - *Masová migrace* - více procesů si vybere stejný uzel, ten se stane přetíženým

- *Oscilace* - proces se může pohybovat po síti, aniž by něco počítal
    - řešení - zavedení kreditů, od určitého počtu může migrovat
- *Zbytečná migrace*

Přiřazení procesů na jednotlivé uzly

- **Alokování uzlů**
    - 1 proces na 1 uzel
        - Např. pevně daná u paralelního počítače
        - 1 proces dokáže plně využít celý uzel, takže nemá smysl jich na jednom uzlu spouštět několik
        - OS uzlu neumí spustit více jak jeden proces najednou
    - Potenciálně nula až několik procesů na jeden uzel
    - Přidělení celé sítě pro jeden výpočet
        - Celkový čas výpočtu je pak dán
            - Dobou k zavedení programů, spuštění procesů a distribuce dat do uzlů
            - Vlastním výpočetem
            - Získáním výsledků z uzlů
    - Přidělení části sítě jednomu výpočtu
    - Několik paralelně běžících výpočtů
        - Na jednom uzlu může běžet několik procesů
        - Nelze se spoléhat na odvozená urychlení, protože ta nepočítala se zátěží, kterou vygeneruje neznámý kód
        - Nehodí se pro synchronní/lockCstepped algoritmy – na společném uzlu by dva spolupracující procesy na sebe musely čekat dobu výpočtu jednoho kroku
- **Identifikace procesů**
    - Jedinečná ID procesů
    - Interakce send/receive (vše ostatní je na nich postaveno)
    - Podle přidělení na uzly:
        - 1 uzel – 1 proces
        - Více procesů na uzlu
        - Více procesů na uzlu a procesy mohou migrovat (tabulka umístění procesů)

Překladače – typy, struktura a princip činnosti

Wednesday, May 29, 2013 4:50 PM

Formálně je překladač **zobrazením ze zdrojového jazyka do cílového jazyka**.

Typy

Postup při tvorbě cílového spustitelného programu:

Zdrojový program → [preprocesor] → upravený zdrojový program → [kompilátor]→cílový program v jazyce symbolických instrukcí → [assembler] → relokovatelný strojový kód & zvenku: knihovní soubory a další relokovatelné objektové soubory → [linker/loader] → cílový strojový kód

Preprocesory

Realizují vnořování částí programu do hostitelského jazyka. Např. expandují makra, přidají include <něco.h> apod. Jeho úkolem je posbírat zdrojový program.

Kompilátory

Generují z vyššího programovacího jazyka kód – strojový/symbolický/jiný jazyk (1. Fortran IBM, 50. léta). Assembly language (jazyk symbolických adres) je jednodušší vyplivnout jako výstup a je i jednodušší pro debugování.

Interprety

Namísto přeložení celého programu provádí příkaz za příkazem operace uvedené ve zdrojovém programu nad vstupními daty. Proto jsou interprety obecně pomalejší. Obvykle ale díky spouštění programu příkaz za příkazem lépe diagnostikují chyby než kompilátory.

Java – kombinace kompilace a interpretace

Zdrojový program je nejříve zkompilován do *bytecode* a ten je pak interpretován virtuálním strojem.

Výhoda – bytecode může být zkompilován na jednom stroji a interpretován na jiném. Aby to bylo rychlejší, některé Java kompilátory (*just-in-time* kompilátory) převádí bytecode do strojového jazyka těsně předtím, než proběhne intermediate (vnitřní) program pro zpracování vstupních dat.

Zdrojový program → [Translator] → intermediate program + vstup → [Virtual Machine] → výstup

Assemblery

Překládají z jazyka symbolických instrukcí (JSI) do strojového kódu. Přeložený strojový kód může být buďto *absolutní binární kód* nebo *přenositelný binární kód.*

Hlavní problémy, které řeší, je adresace symbolických jmen a makra.

Linker a loader

Větší programy jsou často kompilované po částech, takže vytvořený relokovatelný strojový kód (= lze jej umístit do libovolného místa v paměti) je potřeba spojit s dalšími relok. objektovými soubory a knihovními soubory. O to se stará **Linker.** Řeší adresy externí paměti, kde kód v jednom souboru může odkazovat na místo v jiném souboru. **Loader** pak narve všechny spustitelné objektové soubory do paměti pro spuštění.

Typy překladačů

Formátory textu

jde o úpravu textu podle požadavků uživatele, např. TeX. Např. syntax highlighting, nebo přeformátování kódu – odsazení apod. Takový překladač, který ze vstupního souboru definovaného určitým jazykem vygeneruje výstup.

Silikonový překladač

Pro návrh integrovaných obvodů. Proměnné nereprezentují místo v paměti, ale logickou proměnnou obvodu. Výstupem je návrh obvodu.

Dávkový překladač

Dávkové zpracování

Inkrementální překladač
Je interaktivní a překládá po úsecích

Křížový překladač
Překládá na jiném procesoru než na kterém se program (přeložený kód) spouští (např. zabudované – embedded – systémy)

Kaskádní překladač
Máme již překlad z jazyka A do jazyka B, chceme ale A→C. Pak vytvoříme kompilátor z jazyka B do jazyka C, pokud je to snazší než vytvořit kompilátor z jazyka A do jazyka C. Jazyk B je vnitřním jazykem, a pokud je to standardní všeobecně používaný jazyk, pak programy v jazyce A budou snadno přenositelné.

Nevýhodou je však, že oba překladače produkují chybové zprávy. Chybové zprávy druhého překladače jsou cizí pro uživatele jazyka A, protože jsou orientovány na jazyk B. Chybová hlášení výpočtu tak budou pomíchaná.

Paralelizující překladač
Zjišťuje nezávislost úseků programu

Optimalizující překladač
Možnosti ovlivnění optimalizace času či paměti programátorem.

Konverzační překladač - interaktivní

Struktura, princip činnosti
Překladače jsou dva druhy: kompilátory a interprety

Struktura Kompilátoru
všechny příkazy překládá najednou, program lze spustit až po ukončení celého překladu (Pascal, C, Fortran, Ada, …)

ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **generování kódu**, cílový program

Všechny části spolupracují s pracovními tabulkami překladače. Základní tabulkou kompilátoru i interpretu je **tabulka symbolů.** Obsahuje záznamy o názvech proměnných, jejich typu, rozsahu, názvy procedur společně s věcmi jako počet a typy argumentů, metoda předání jednotlivých argumentů (odkazem nebo hodnotou) a návratový typ.

Výhodou kompilátoru je rychlá exekuce programu.

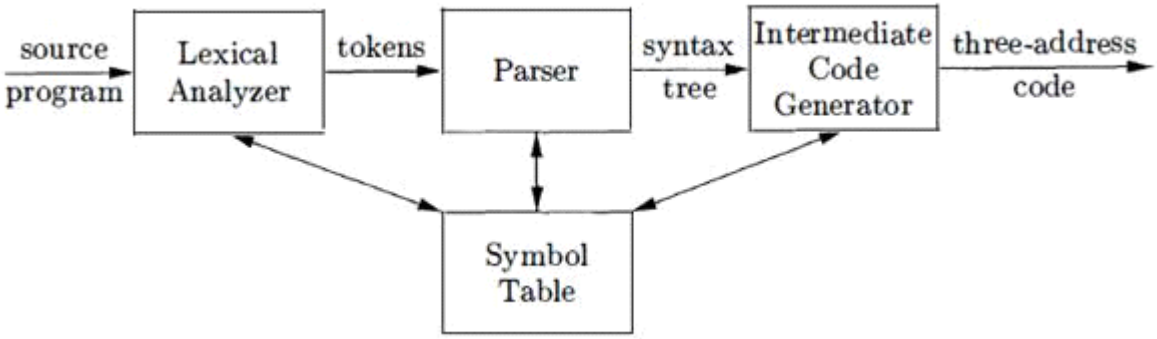
Struktura Interpreta
zpracovává příkazy jednotlivě a každý provede okamžitě po jeho přeložení (Python, Perl, JavaScript, Ruby, …)

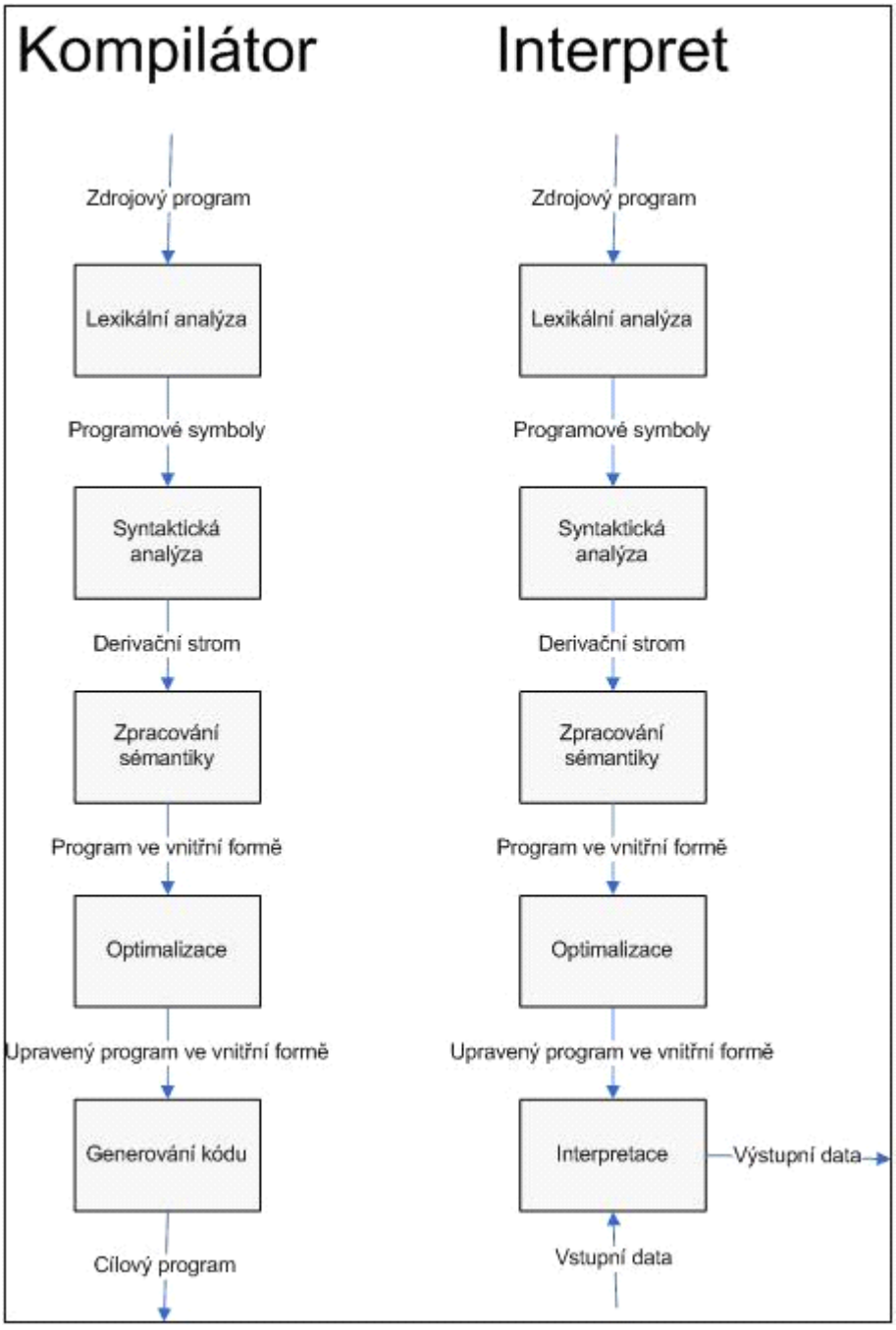
ANALÝZA: zdrojový program → **lexikální analýza** (lineární), programové symboly → **syntaktická analýza** (hierarchická), derivační strom

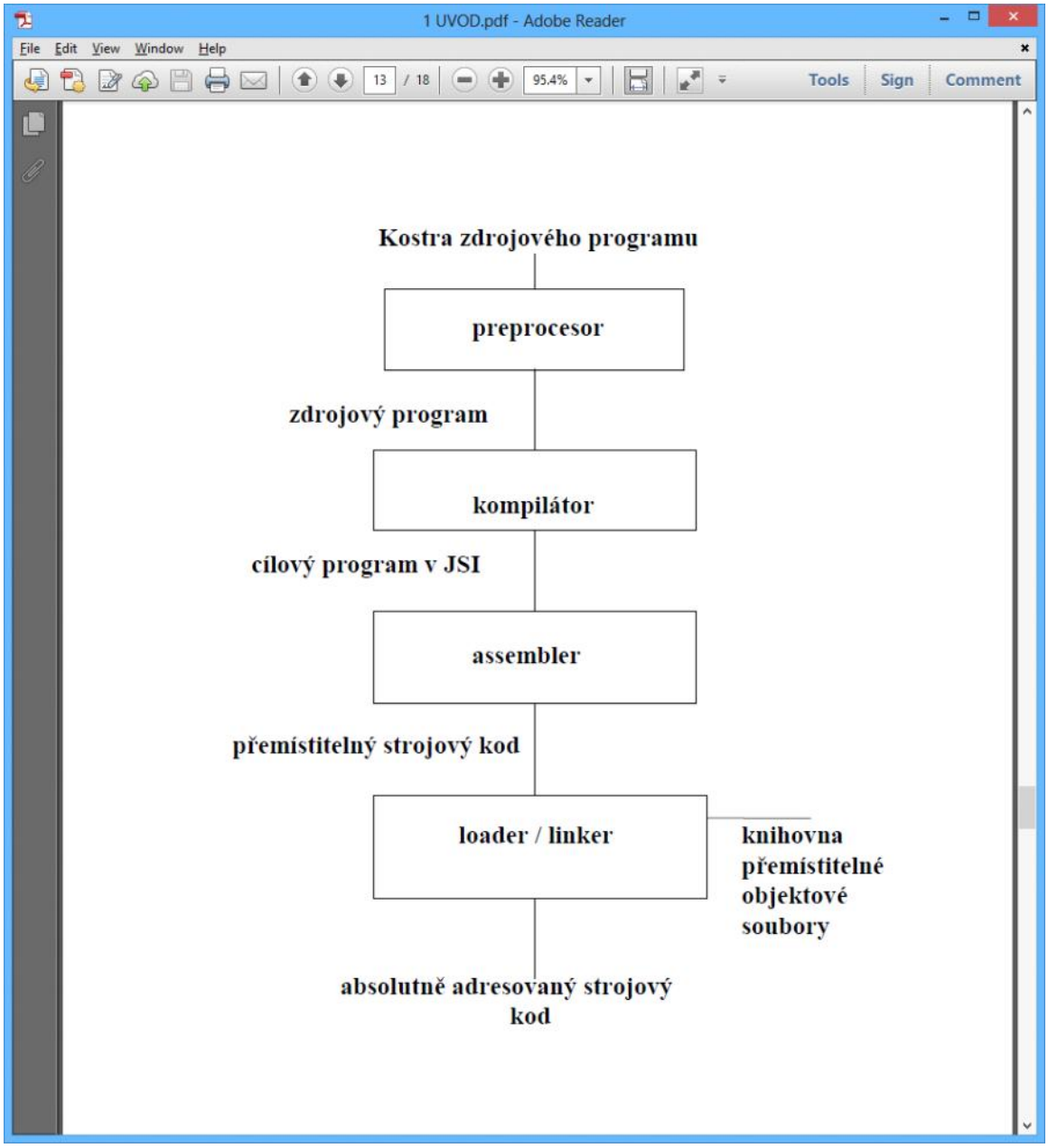
SYNTÉZA: derivační strom → **zpracování sémantiky**, program ve vnitřní formě → **optimalizace** (příprava generování), upravený program ve vnitřní formě → **interpretace**, pracuje se vstupními daty, aby vygeneroval výstupní data

Výhodou interpretu je:

- Eliminace kroků cyklu „editace → překlad → sestavení → exekuce“
- Snazší realizace ladících mechanismů (zachování původních jmen symbolů)


source program
Lexical Analyzer
tokens
Parser
syntax tree
Intermediate Code Generator
three-address code
Symbol Table


Kompilátor
Interpret
Zdrojový program
Lexikální analýza
Programové symboly
Syntaktická analýza
Derivační strom
Zpracování sémantiky
Program ve vnitřní formě
Optimalizace
Upravený program ve vnitřní formě
Generování kódu
Cílový program
Zdrojový program
Lexikální analýza
Programové symboly
Syntaktická analýza
Derivační strom
Zpracování sémantiky
Program ve vnitřní formě
Optimalizace
Upravený program ve vnitřní formě
Interpretace
Výstupní data
Vstupní data

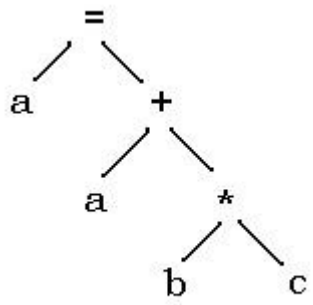


Lexikální analýza

zdrojový kód vstupuje do procesu překladu jako posloupnost znaků. Tato posloupnost se čte lineárně zleva doprava a sestavují se z ní *lexikální symboly* jako konstanty, identifikátory, klíčová slova nebo operátory. Je založena na regulárních gramatikách. Výsledkem je posloupnost symbolů, např. je na vstupu rozeznáno klíčové slovo begin a do posloupnosti lexikálních symbolů bude zařazen nový symbol reprezentující právě toto klíčové slovo. Tyto symboly jsou programem snadno použitelné a dále zpracovatelné. V této fázi se odstraňují veškeré komentáře.

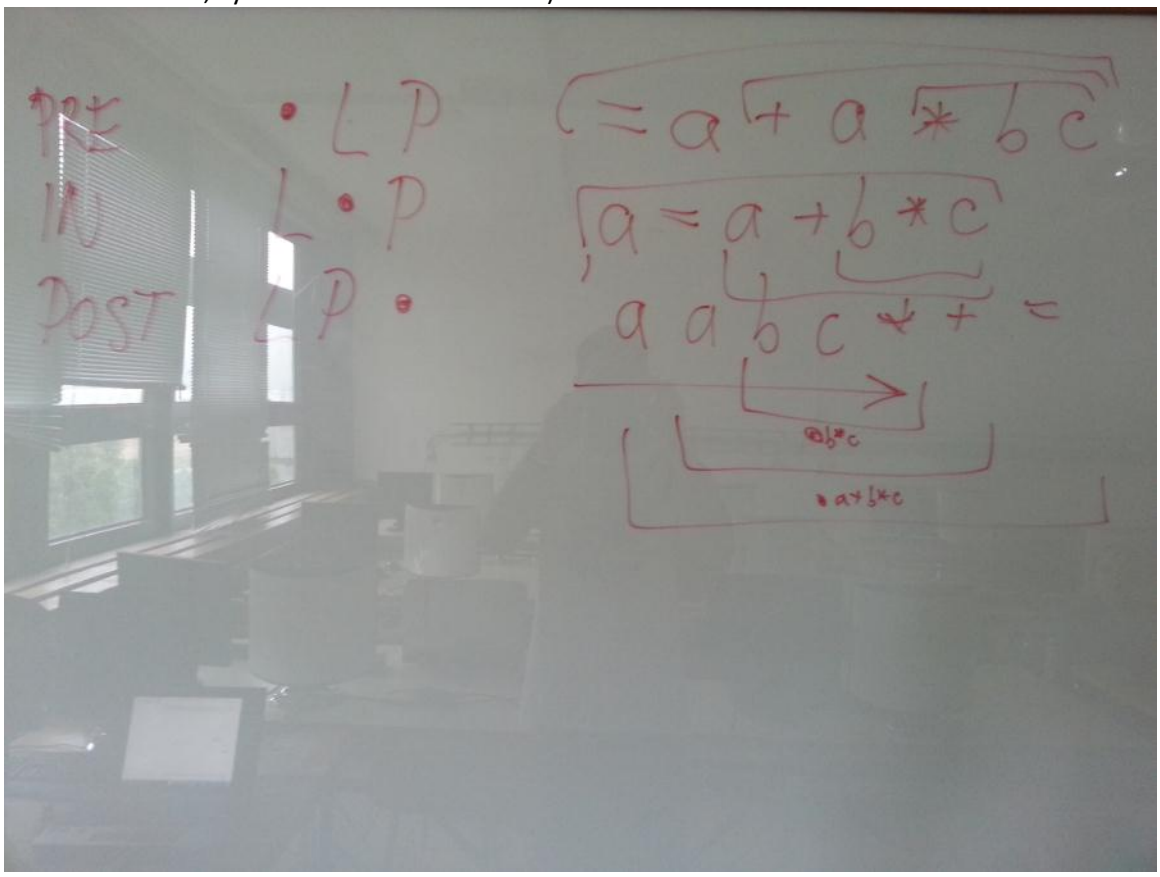
Syntaktická analýza

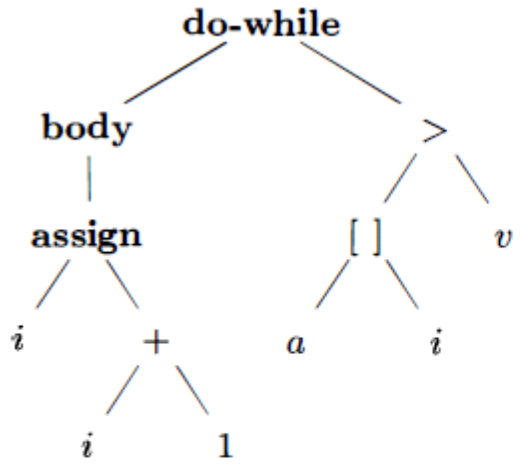
Z posloupnosti lexikálních symbolů se vytvářejí hierarchicky zanořené struktury (vnitřní jazyk překladače), které mají jako celek svůj vlastní význam, např. výrazy, příkazy, deklarace nebo program. Programy jsou psány většinou v infixové notaci (a=a+b*c) => analyzujeme a vytváříme hierarchické uspořádání derivačního stromu:



Notace vnitřního jazyka překladače:

- Prefixová (nemá závorky, operátory bezprostředně *předchází* operandy a pořadí operandů je zachováno)
- Infixová
- Postfixová (nemá závorky, operátory bezprostředně *následují* operandy a pořadí operandů je zachováno, vyhodnotitelná zásobníkem)





(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v);`"

Sémantická analýza

Provádějí se některé kontroly, zajišťující správnost programu z hlediska vazeb, které nelze provádět v rámci syntaktické analýzy (některé konstrukty nejdou popsat bezkontextovou gramatikou, třeba např. kontrola deklarací, typová kontrola, kontrola, jestli index pole je integer apod.).

Typická reprezentace programu ve *vnitřní formě* (intermediate code) je sekvence trojic nebo čtveřic (3- nebo 4-adresových instrukcí)

Optimalizace

Optimalizátor kódu zajišťuje, aby se používalo co nejméně pomocných proměnných pro mezivýpočty, aby se v cyklu zbytečně několikrát nevyhodnocoval tentýž výraz, jestliže hodnota jeho prvků zůstává bez změny a vyhodnocení stačí provést jednou před cyklem, apod. Optimalizací prochází program obvykle v intermediálním tvaru – intermediální kód je již podobný cílovému programu, má však strukturu vhodnější pro optimalizaci. Může to být zápis podobný assembleru nebo třeba dynamická struktura (dynamický seznam stromů představujících jednotlivé příkazy).

Generování kódu

poslední fází překladače je generování cílového kódu, což je obvykle přemístitelný kód nebo program jazyka asembleru. Všem proměnným použitým v programu se přidělí místo v paměti. Potom se instrukce mezikódu překládají do posloupnosti strojových instrukcí, které provádějí stejnou činnost.

Vícefázový a víceprůchodový překladač

Fáze = logicky dekomponovaná část (může obsahovat více průchodů, např. optimalizace)

Průchod = čtení vstupního řetězce, zpracování, zápis výstupního řetězce – může obsahovat více fází

Jednoprůchodový překladač = všechny fáze probíhají v rámci jediného čtení zdrojového textu programu

- Omezená možnost kontextových kontrol
- Omezená možnost optimalizace
- Lepší možnosti zpracování chyb a ladění (tedy dobré pro výuku)

Na strukturu překladače mají vliv:

- Vlastnosti zdrojového a cílového jazyka
- Vlastnosti hostitelského počítače

- Rychlost/velikost překladače
- Rychlost/velikost cílového kódu
- Ladicí schopnosti (detekce chyb, zotavení)
- Velikost projektu, prostředky, termíny

Testování a údržba překladače

Díky formální specifikaci jazyka je možné automatické provádění testů

Systematického testování lze dosáhnout regresními testy, což je sada testů doplňovaná o testy na odhalené chyby. Po každé změně v překladači se provedou všechny testy a jejich výstupy se porovnají s předešlými.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Regulární gramatiky, regulární výrazy a konečné automaty

Thursday, May 30, 2013 8:36 AM

Regulární gramatiky

Gramatika G je čtveřice (N,Σ,P,S), kde:

- N je konečná množina neterminálních symbolů (neterminálů).
- Σ je konečná množina terminálních symbolů tak, že žádný symbol nepatří do N a Σ zároveň (jsou disjunktní).
- P je konečná množina odvozovacích pravidel. Každé pravidlo je tvaru

$$(\Sigma \cup N)^* \longrightarrow (\Sigma \cup N)^*$$

"cokoli poskládaný ze všech možnejch symbolů na cokoli"; S je prvek z N nazývaný počáteční symbol.

```
- VĚTNÁ FORMA
    Def.: Řetězec α se nazývá větnou formou  v gramatice G,
        s počátečním symbolem S, platí-li:

          S  =>* α, kde          α  ∈ (N ∪ T)*


- VĚTA
    Def.: Řetězec α se nazývá větou v gramatice G,
        s počátečním symbolem S, platí-li:

          S  =>* α, kde          α  ∈ T*


- FRÁZE
        Def.: Nechť λ =  α β γ je větná forma v gramatice G.
    Podřetězec β se nazývá frází větné formy λ vzhledem
    k neterminálnímu symbolu A, platí-li
              S  =>* α A γ     a     A =>* β

    Tzn.frázi tvoří listy podstromu derivačního stromu.

-   JEDNODUCHÁ FRÁZE větné formy α A γ  vzhledem k neterm. A je
    podřetězec β, platí-li
              S  =>* α A γ     a     A => β

-   L-FRÁZE
              je nejlevější jednoduchou frází
```

Lineární gramatika = bezkontextová gramatika, která má nanejvýš jeden neterminál na pravé straně. Regulární gramatika je speciálním případem lineární gramatiky, kdy všechny neterminály jsou na levém konci/straně (levá lineární = levá regulární) nebo ekvivalentně pro pravou stranu.

Regulární gramatika – je to gramatika typu 3 = lineární, navíc převedená do regulárního tvaru (podle Chomského hierarchie). Pravidla těchto lineárních gramatik jsou omezena na jeden neterminál na levé straně. Pravá strana se u pravé regulární gramatiky skládá z jednoho terminálu (u lineární i z více), který může být následován jedním neterminálem, tedy:

$$X \to wY$$
$$X \to w,$$

kde X, Y jsou neterminály a w je řetězcem terminálů. Regulární gramatiky se také nazývají **pravé lineární gramatiky**. Obdobně se definují i **levé regulární gramatiky**, které obsahují pravidla typu:

$$X \to Yw$$
$$X \to w$$

Pravé a levé gramatiky jsou ekvivalentní. Jazyky generované regulárními (=lineárními) gramatikami jsou právě jazyky rozpoznatelné konečným automatem.

Lineární gramatika = má na pravé straně právě jeden neterminál

Regulární gramatika = gramatika, která popisuje regulární jazyk, přesně definovaný tvar pravidel (B -> a, B -> aC, B -> e pro pravou regulární gramatiku)

Regulární gramatika je tedy buď jen levá lineární gramatika nebo jen pravá lineární gramatika. Čistě lineární (levo-pravá) gramatika je pak taková gramatika, která sestává z pravých i levých pravidel současně.

Regular languages are also characterized by special grammars called regular grammars whose productions take the following form, where w is a string of terminals.
A → wB or A → w.
*Example. A regular grammar for the language of a*b* is*
S → Λ | aS | T
T → b | bT.
http://www.postech.ac.kr/~seungjin/courses/automata/handouts/handout04_4pp.pdf

http://web.cecs.pdx.edu/~jhein/lectures/Section.11.4.pdf

Grammar	Languages	Automaton	Production rules (constraints)
Type-0	Recursively enumerable	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type-2	Context-free	Non-deterministic pushdown automaton	$A \rightarrow \gamma$
Type-3	Regular	Finite state automaton	$A \rightarrow a$ and $A \rightarrow aB$

From <http://en.wikipedia.org/wiki/Chomsky_hierarchy>

Regulární výrazy

Regulární výrazy umožňují algebraické manipulace s regulárními množinami - umožňují **vyjádření regulárních množin**. Třída regulárních výrazů nad abecedou Σ je definována takto:

- *e* a ∅ jsou regulární výrazy
- každé písmeno (symbol - znak) $\sigma \in \Sigma$ je regulární výraz nad Σ
- jsou-li R_1 a R_2 regulární výrazy nad Σ, pak i $(R_1 + R_2)$, $(R_1 . R_2)$ a R_1^* jsou regulární výrazy nad Σ

Daná množina je regulární množina nad Σ, právě když může být popsána vhodným regulárním výrazem nad Σ. Každý regulární výraz **U** popisuje jistou množinu **Ũ** slov nad **Σ**: **Ũ** ⊆ **Σ***

Regulární množiny se vhodně charakterizují přechodovými grafy. Přechodový graf T nad abecedou Σ je konečný orientovaný graf, jehož každá hrana je pojmenována jistým slovem $w \in \Sigma^*$;alespoň jeden uzel je počáteční.

Množinu všech slov akceptovaných konečným automatem **A** označíme **Ã**. Množina je regulární nad **Σ** právě když je akceptována vhodným automatem nad **Σ**

Regulární výraz je řetězec popisující celou množinu řetězců (slov), konkrétně regulární jazyk. Používají se nejčastěji v počítačových programech a skriptovacích jazycích pro vyhledávání a úpravu textu. V případě, že uživatel chce v textu vyhledat nějaký řetězec, který nezná přesně nebo který může mít více variant, může zadat regulární výraz, který postihne všechny chtěné varianty. Program tak nalezne všechny části textu, které danému výrazu odpovídají.

Každý z regulárních výrazů označuje jistý regulární jazyk.

Kleeneho teorém: Každý regulární výraz je převoditelný na konečný automat.

Konečné automaty

formálně je konečný automat definován jako **uspořádaná pětice (S,Σ,P,s,F)** , kde:

S je konečná množina stavů.

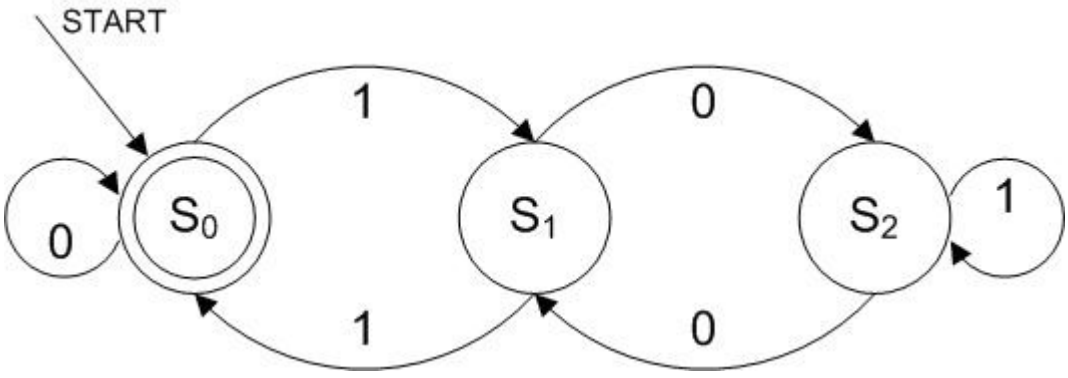
Σ (*velké sigma*) je konečná množina vstupních symbolů nazývaná abeceda.

P je tzv. přechodová funkce (též přechodová tabulka), formálně zobrazení $\delta: S \times \Sigma \rightarrow S$, popisující pravidla přechodů mezi stavy. Přechod je určen stavem ve kterém se automat nachází a symbolem, který přichází na vstup (nebo který je čten na vstupu)

s je počáteční stav (s naleží S)

F je množina koncových (přijímacích) stavů (F je podmnožinou S)

Popis činnosti automatu: Na počátku se automat nachází v definovaném počátečním stavu. Dále v každém kroku přečte jeden symbol ze vstupu a přejde do stavu, který je dán hodnotou, která v přechodové tabulce odpovídá aktuálnímu stavu a přečtenému symbolu. Poté pokračuje čtením dalšího symbolu ze vstupu, dalším přechodem podle přechodové tabulky, atd. Podle toho, zda automat skončí po přečtení vstupu ve stavu, který patří do množiny koncových stavů, platí, že automat buď daný vstup přijal nebo nepřijal. Množina všech řetězců, který daný automat přijme, tvoří regulární jazyk.



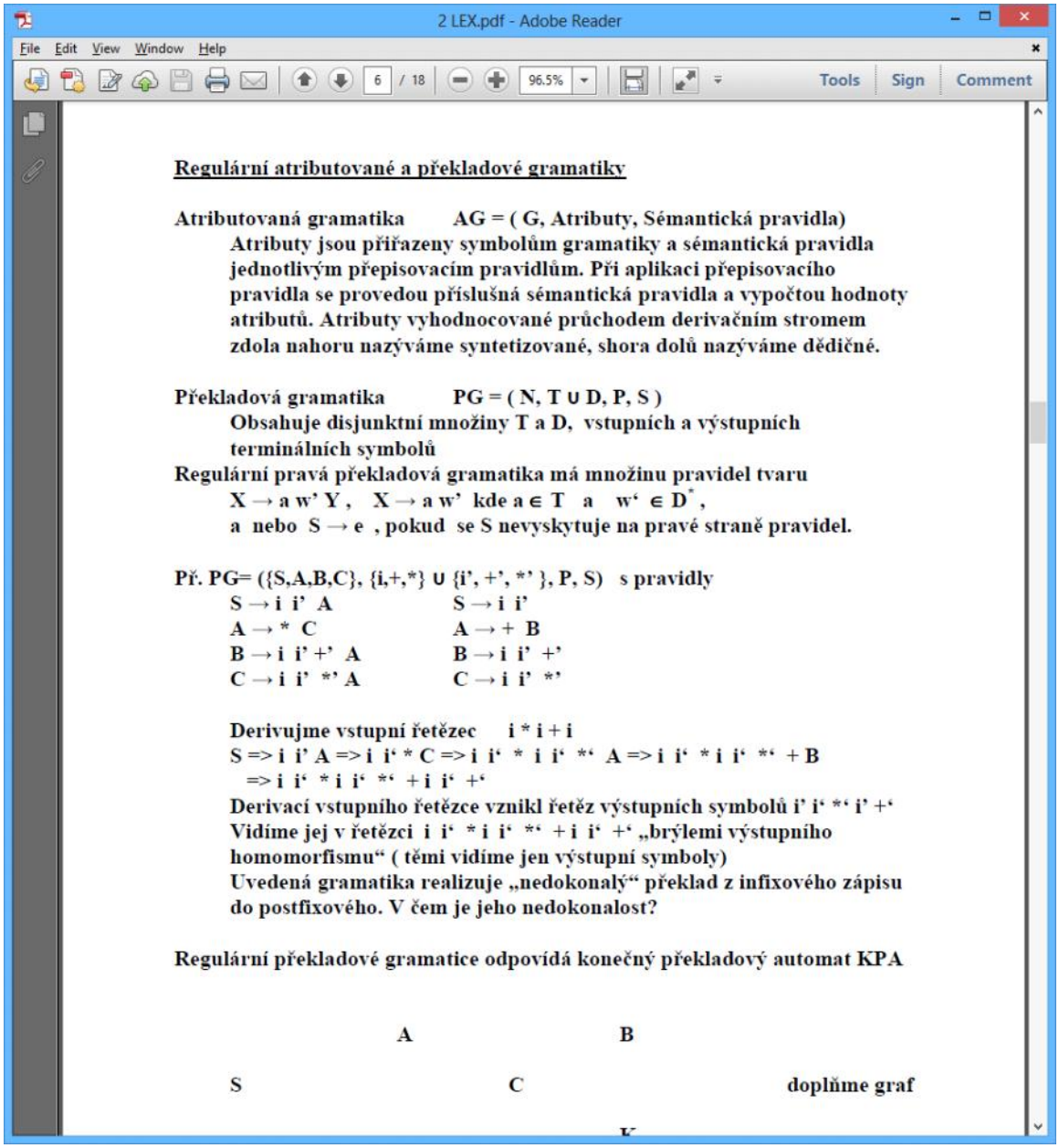
Determinismus

Konečné automaty se dělí dále na deterministické (DKA, DFA) a nedeterministické (NKA, NFA). U deterministických automatů, každý stav má právě jeden možný přechod pro každý možný vstup. U nedeterministického automatu jsou navíc povoleny e-hrany a více hran z jednoho stavu pro stejný vstup => v jeden okamžik se NKA může nacházet ve více stavech současně. Existuje však algoritmus, který umožňuje libovolný NKA převést na o něco složitější DKA (nejhůře v exponenciálním čase, prakticky však řádově rychleji).

Meze regulárních gramatik

Jak určit zdali je nějaký jazyk možné rozpoznat regulárním výrazem (konečným automatem, regulární gramatikou)? K tomuto lze použít tzv. Pumping teorém [wiki] nebo česky [abclinuxu].

Teorém vlastně říká, že v dostatečně dlouhém slově *w* daného regulárního jazyka můžeme nalézt tři části — *x*, *y* a *z*, přičemž nejdůležitější část *y* může zahrnovat i celé slovo. Aby byl tento jazyk regulární, musí platit, že část *y* můžeme ze slova vyjmout, nebo jí libovolně zopakovat, a přitom stále zůstáváme v rámci stejného jazyka.



Regulární atributované a překladové gramatiky

Atributovaná gramatika AG = (G, Atributy, Sémantická pravidla)

Atributy jsou přiřazeny symbolům gramatiky a sémantická pravidla jednotlivým přepisovacím pravidlům. Při aplikaci přepisovacího pravidla se provedou příslušná sémantická pravidla a vypočtou hodnoty atributů. Atributy vyhodnocované průchodem derivačním stromem zdola nahoru nazýváme syntetizované, shora dolů nazýváme dědičné.

Překladová gramatika PG = (N, T ∪ D, P, S)

Obsahuje disjunktní množiny T a D, vstupních a výstupních terminálních symbolů

Regulární pravá překladová gramatika má množinu pravidel tvaru

$X \rightarrow a\ w'\ Y$, $X \rightarrow a\ w'$ kde $a \in T$ a $w' \in D^*$,

a nebo $S \rightarrow e$, pokud se S nevyskytuje na pravé straně pravidel.

Př. PG= ({S,A,B,C}, {i,+,*} ∪ {i', +', *' }, P, S) s pravidly

$S \rightarrow i\ i'\ A$	$S \rightarrow i\ i'$
$A \rightarrow *\ C$	$A \rightarrow +\ B$
$B \rightarrow i\ i'\ +'\ A$	$B \rightarrow i\ i'\ +'$
$C \rightarrow i\ i'\ *'\ A$	$C \rightarrow i\ i'\ *'$

Derivujme vstupní řetězec i * i + i

S => i i' A => i i' * C => i i' * i i' *' A => i i' * i i' *' + B

=> i i' * i i' *' + i i' +'

Derivací vstupního řetězce vznikl řetěz výstupních symbolů i' i' *' i' +'

Vidíme jej v řetězci i i' * i i' *' + i i' +' „brýlemi výstupního homomorfismu" (těmi vidíme jen výstupní symboly)

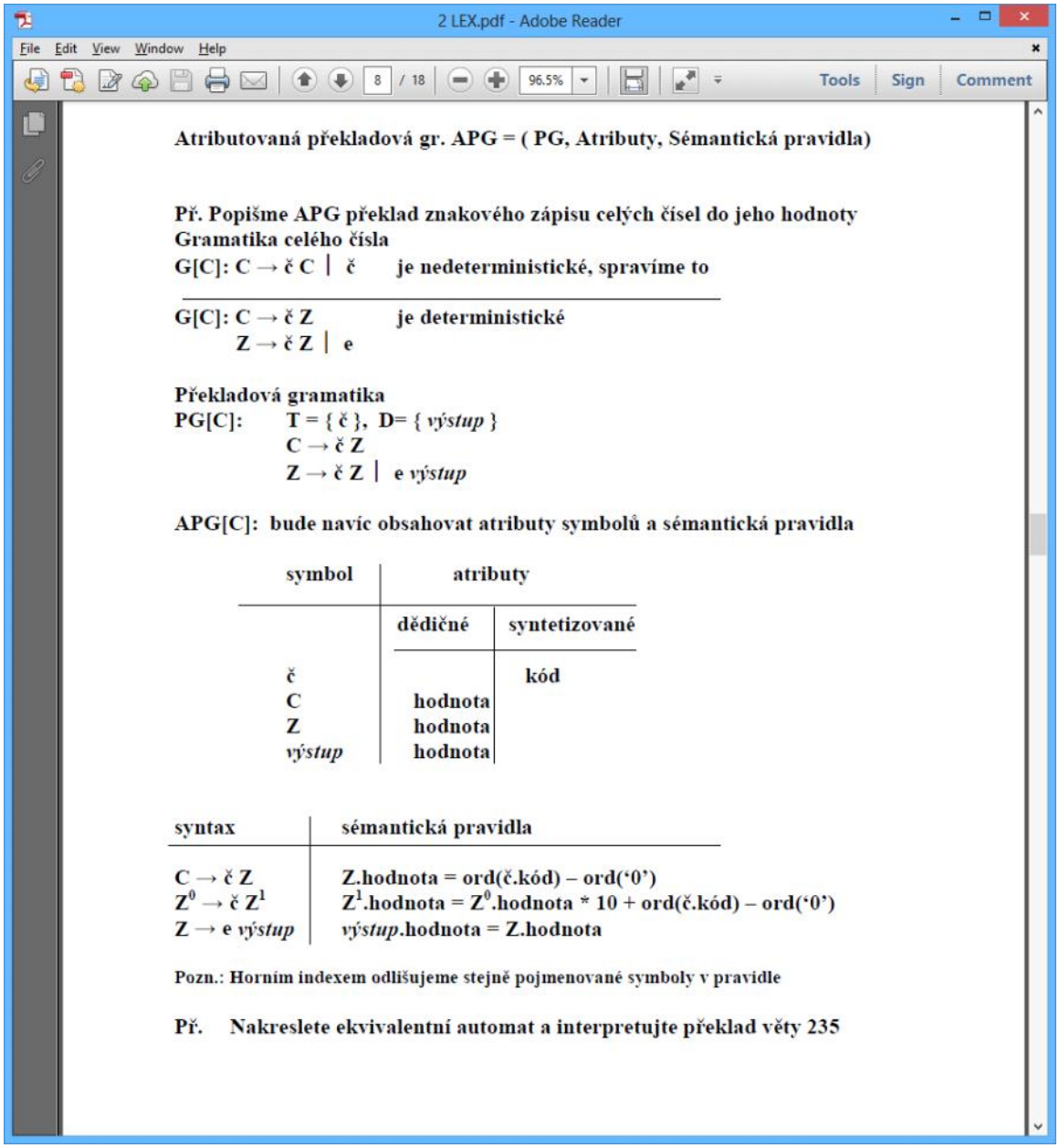
Uvedená gramatika realizuje „nedokonalý" překlad z infixového zápisu do postfixového. V čem je jeho nedokonalost?

Regulární překladové gramatice odpovídá konečný překladový automat KPA

A B

S C doplňme graf

K



Atributovaná překladová gr. APG = (PG, Atributy, Sémantická pravidla)

Př. Popišme APG překlad znakového zápisu celých čísel do jeho hodnoty
Gramatika celého čísla
G[C]: C → č C | č **je nedeterministické, spravíme to**

G[C]: C → č Z **je deterministické**
Z → č Z | e

Překladová gramatika
PG[C]: **T = { č }, D= { *výstup* }**
C → č Z
Z → č Z | e *výstup*

APG[C]: bude navíc obsahovat atributy symbolů a sémantická pravidla

symbol	atributy	
	dědičné	syntetizované
č		kód
C	hodnota	
Z	hodnota	
výstup	hodnota	

syntax	sémantická pravidla
$C \rightarrow č\ Z$	Z.hodnota = ord(č.kód) – ord('0')
$Z^0 \rightarrow č\ Z^1$	Z^1.hodnota = Z^0.hodnota * 10 + ord(č.kód) – ord('0')
Z → e *výstup*	*výstup*.hodnota = Z.hodnota

Pozn.: Horním indexem odlišujeme stejně pojmenované symboly v pravidle

Př. Nakreslete ekvivalentní automat a interpretujte překlad věty 235

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Ekvivalence konečných automatů a regulárních gramatik

Thursday, May 30, 2013 8:38 AM

Regulární gramatiky popisují všechny *regulární jazyky* a v tomto smyslu (ve schopnosti popisu jazyka) jsou ekvivalentní s konečnými automaty a regulárními výrazy. Regulární gramatika ja buď pravá regulární (neterminály jsou vpravo) nebo levá regulární (neterminály jsou vlevo).

Regulární jazyk je formální jazyk (množina (i nekonečná) slov složených z omezené abecedy), který:

- může být akceptován deterministickým/nedeterministickým konečným stavovým automatem
- lze popsat regulárním výrazem
- lze ho generovat regulární gramatikou

Příkladem **ne**regulárního jazyka je $a^n b^n$, kde n > 1 (alespoň jedno *a* následované stejným počtem *b*), gramatika pro palindromy apod. - *lze určit na základě Nerodovy věty, která se užívá v důkazech, že nějaký jazyk není rozpoznatelný konečným automatem*

Každý regulární jazyk je rozpoznatelný konečným automatem; každý jazyk rozpoznatelný konečným automatem je regulární.

Kleenova věta: Libovolný jazyk je regulární, právě když je rozpoznatelný konečným automatem. Přechodový graf je T nad **S** je konečný orientovaný graf, jehož každá hrana je pojmenována jistým slovem w∈**S***. Alespoň jeden z uzlů grafu je počáteční a některé uzly jsou koncové. Ke každému přechodovému grafu T nad abecedou S existuje regulární výraz R nad S takový, že

$$\tilde{\tilde{R}} = \tilde{\tilde{T}}$$

a ke každému regulárnímu výrazu R nad S existuje konečný automat A takový, že

$$\tilde{\tilde{A}} = \tilde{\tilde{R}}$$

.

Postup převodu gramatiky na konečný automat

Potřebujeme získat gramatiku typu 3 ve standardní formě.

Regulární gramatika je ve **standardní formě**, jestliže obsahuje pouze pravidla tvaru $X \rightarrow aY$ a $X \rightarrow a$, $X \rightarrow e$ kde *X,Y* jsou neterminály, *a* je právě jeden terminál, *e* je prázdný symbol. Toho dosáhneme takto:

- Původní gramatika typu 3 (lineární): G = (N, T, S, P)
- Požadovaná regulární gramatika: G' = (N', T, S, P')
- Požadovaná gramatika G' bude mít stejné terminální symboly a stejný počáteční stav.
- Konstrukce přechodů *P'*:
  - do *P'* zařadíme všechna pravidla z *P* ve tvaru $X \rightarrow aY$ a $X \rightarrow e$
  - za každé pravidlo $X \rightarrow x1\ x2\ x3\ Y$ zařadíme do *P'* soustavu pravidel:
    - $X \rightarrow x1\ X1$
    - $X1 \rightarrow x2\ X2$
    - $X2 \rightarrow x3\ Y$
      - za každé pravidlo $X \rightarrow z1\ z2$, zařadíme do *P'* soustavu:
    - $X \rightarrow z1\ Z1$
    - $Z1 \rightarrow z2\ Z2$
    - $Z2 \rightarrow e$

- Místo pravidel tvaru $X \rightarrow Y$ musíme zajistit to, aby z každého stavu X pro který máme $X \rightarrow Y$, bylo možné odvodit všechny řetězce, které lze odvodit z Y.
- N' vznikne obohacením N o všechny nově vytvořené neterminální symboly

Zkonstruujeme automat z nově vytvořené gramatiky

- stavy budou odpovídat neterminálním symbolům
- vstupy budou odpovídat terminálním symbolům
- přechodovou funkci zkonstruujeme na základě analogií
  - $X \rightarrow aY \leftrightarrow$ přechod ze stavu X do stavu Y při vstupu symbolu a
- počáteční stav bude odpovídat počátečnímu symbolu
- množinu koncových stavů určíme z pravidel $X \rightarrow e$

Tímto jsme získali **nedeterministický konečný automat**, který lze převést na **deterministický konečný automat.**

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Nedeterministický a deterministický konečný automat

Thursday, May 30, 2013 8:38 AM

Deterministický konečný automat

Je uspořádaná pětice (S,Σ,P,s,F), kde:

- S je konečná množina stavů.
- Σ je konečná množina vstupních symbolů nazývaná abeceda.
- P je tzv. přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy.
- s je počáteční stav (s naleží S)
- F je množina koncových stavů (F je podmnožinou S)

Nedeterministický konečný automat

Nedeterministickým konečným automatem (NKA) *bez výstupu* nazýváme každou pětici

A = (Q, Σ, δ, S, F), kde:

- Q je konečná, neprázdná, množina stavů
- Σ je konečná neprázdná množina vstupních symbolů (vstupní abeceda)
- δ (přechodová funkce) je zobrazení δ: QxΣ → P(Q). Kde P(Q) je potenční množina (množina všech podmnožin množiny Q včetně prázné množiny *e*)
- S je množina počátečních stavů (S náleží Q) - není jednoznačně určen počáteční stav
- F je množina koncových stavů (F náleží Q)

Oborem hodnot přechodové funkce jsou všechny podmnožiny množiny stavů

Formálně je definován podobně jako DKA, ale obsahuje prvky nedeterminismu:

1. nejednoznačně určený počáteční stav (může jich být více)
2. nejednoznačné přechody (při přijetí stejného vstupu lze přejít do více stavů)
3. e - přechody (přechod do stavu bez přijetí vstupního symbolu)

- chování NKA lze popsat sekvencí pozic (množina stavů, ve kterých se automat může nacházet), z nichž každá jednoznačně definuje, zda je zpracovaný řetězec akceptován či zamítnut
- pozic je konečný počet
- přechody mezi pozicemi jsou jednoznačné
- Nejdůležitější rozdíl mezi DKA a NKA je v tom, že výsledkem přechodové funkce není pouze jeden stav, ale množina stavů, která může být i prázdná
- to vše jsou vlastnosti DKA a proto **ke každému NKA existuje ekvivalentní DKA**

V případě nedeterministického konečného automatu (NKA) je vstupní slovo akceptováno (rozpoznáno,) pokud toto slovo může automat převést do některého z koncových stavů (množina F) z některého z počátečních stavů (množina S).

Převod NKA na DKA

1. Lineární gramatiku nejprve převedeme na regulární tvar (postup viz otázka [*Ekvivalence konečných automatů a regulárních gramatik*].

   2.Pak zkonstruujeme nedeterministický konečný automat a z něho nakonec deterministický (jak viz dále).

   A) Hlavní myšlenka je taková, že **každý stav vytvořeného DFA odpovídá množině stavů NFA.**

   B) Nebo: Z nedeterministického automatu se vytváří **strom**, který již popisuje deterministický automat, popisující tentýž problém.

Postup převodu

Samotný převod stojí na myšlence, že pokud lze ze vstupního uzlu

S
přejít jdo uzlu

A
a do uzlu

B
, tak vytvoříme nový uzel, říkejme mu

$[A,B]$
. Tento uzel bude mít stejné vstupy a výstupy jako sjednocení uzlů

A
a

B
. Nyní tabulka převedeného automatu obsahuje dva uzly

$\{S,[A,B]\}$
. Postup opakujeme pro uzel

$[A,B]$
. Takto postupně projdeme všechny stavy nově vytvářeného deterministického automatu.

Koncovými uzly převedeného deterministického automatu budou takové uzly, které jsou nadmnožinou koncových uzlů původního automatu (měl-li původně automat výstupní uzel

A
, tak uzel

$[A,X]$
, který vznikl jako sjednocení uzlu

A
a uzlu

X
, bude také výstupní).

Tento postup zároveň eliminuje všechny stavy, do kterých se deterministická verze automatu nemůže vůbec dostat. Zároveň ale mohou vzniknout uzly, které mají totožné vlastnosti (vstupní a výstupní uzly, konečnost, vlastnost *být počátečním uzlem*). Tyto uzly můžeme po doběhnutí algoritmu ztotožnit.

Příklad

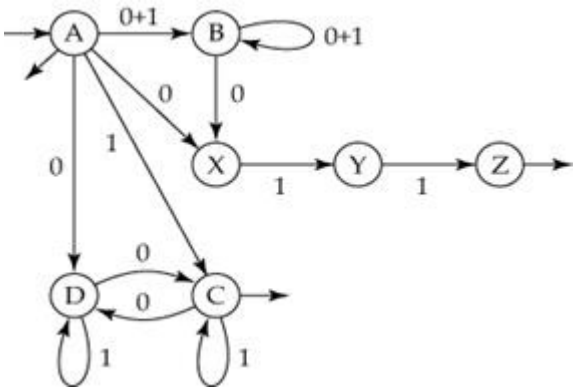
Zadaná pravá lineární gramatika:

```
A --> B | C
B --> 0B | 1B | 011
C --> 0D | 1C | e
D --> 0C | 1D
```

Pravá regulární gramatika:

```
A --> 0B | 1B | 0X | 0D | 1C | e
B --> 0B | 1B | 0X
X --> 1Y
Y --> 1Z
Z --> e
C --> 0D | 1C | e
D --> 0C | 1D
```

Nedeterministický konečný automat:



Deterministický konečný automat:

Přechodovou tabulku deterministického konečného automatu vytvoříme z přechodového diagramu nedeterministického kon. automatu takto:

- Do prvního řádku tabulky zapíšeme počáteční stav automatu a postupně zjistíme, do jakých množin stavů se nedeterministický automat může dostat z tohoto stavu přijmutím jednotlivých symbolů jeho vstupní abecedy.
- Z nalezených množin s více než jedním stavem vytvoříme tzv. *kompozitní* stavy det. automatu. Ty pak použijeme do přechodové tabulky det. automatu jako výstupy přechodové funkce pro počáteční stav a odpovídající vstupní symboly.
- Vzniklé kompozitní stavy (a případně i normální stavy) také využijeme v dalších řádcích přechodové tabulky a případně doplňujeme nové kompozitní stavy, do kterých se můžeme dostat z množin původních stavů každého kompozitního stavu přes vstupní symboly.
- Takto postupně vytvoříme celou přechodovou tabulku ekvivalentního deterministického automatu.
- Kompozitní stavy, zahrnující původní koncové stavy, můžeme označit také jako koncové.

	stav	0	1
<-->	A	BXD	BC
	BXD	BXC	BYD
<--	BC	BXD	BC
<--	BXC	BXD	BYC
	BYD	BXC	BZD
<--	BYC	BXD	BZC
<--	BZD	BXC	BD
<--	BZC	BXD	BC
	BD	BXC	BD

Nové stavy jsou A,BXD, BC, BXC, atd. Přechody 0,1.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Lexikální analýza, princip činnosti

Thursday, May 30, 2013 8:38 AM

Úkoly lexikálního analyzátoru

- Čtení zdrojového textu,
- Nalezení a rozpoznání lexikálních symbolů ve volném formátu textu, včetně případného rozlišení klíčových slov a identifikátorů. Vyžaduje spolupráci se syntaktickým analyzátorem.
- Vynechání mezer a komentářů,
- Interpretace direktiv překladače,
- Uchování informace pro hlášení chyb,
- Zobrazení protokolu o překladu.

Je prováděna **lexikálním analyzátorem**, který je vstupní a nejjednodušší částí překladače. Čte znaky zdrojového programu, a jeho výstupem jsou **tokeny**. Vstupní posloupnost znaků - program - je slučována do lexikologicky smysluplných mnohoznakových jednotek, tzv. ***lexémů*** *(*např. if, foo123bar). Tokeny pak symbolicky reprezentují lexémy (např. *if* pro lexém klíčové slovo *if*, *id* pro identifikátor *foo123bar*) a lexémy jsou tak vlastně jejich instance. Podoba lexémů reprezentujících jednotlivé tokeny je vymezena **vzorem** (**pattern**), typicky regulárním výrazem.
Kromě toho je jeho úkolem odstranění komentářů a eliminace přebytečných bílých znaků.

Token Name	popis (v podstatě pattern)	příklad lexémů	hodnota atributu
if	znaky i, f	if	-
else	zn. e, l, s, e	else	-
id	písmeno násl. písm./číslicí	foo, score, myId	pointer do tab. ~~Záznamů~~ symbolů
number	jakákoliv číselná konstanta	3.14, 10	pointer do tab. záznamů
literal	vše v uvozovkách	“hello world”	pointer do tab. záznamů
relop	<, <=, >, >=, =, <>	<, <=, >, >=, =, <>	LT, LE, GT, GE, EQ, NE

Token je tvořen dvěma částmi – názvem tokenu (token name) a hodnotou atributu (attribute value). Názvy tokenu jsou často abstraktní symboly, které jsou pak použity parserem pro syntaktickou analýzu. Jde např. o nějaké klíčové slovo nebo o soubor znaků představujících identifikátor. Operátory, klíčová slova a další ve skutečnosti atributové hodnoty nepotřebují. Pokud má token hodnotu atributu, jde o pointer do tabulky symbolů, která obsahuje dodatečné informace o tokenu, které nejsou součástí gramatiky.

Proud tokenů je předán parseru pro syntaktickou analýzu. Lexikální analyzátor také obvykle používá tabulku symbolů, do které ukládá objevené lexémy a ze kterých bere informace, aby mohl parseru podstrčit správný token. Jak název tokenu (typ - id, číslo,…), tak jeho atribut (číslo 0/1,…) ovlivňují rozhodování ve fázi parsování a pozdějších fázích. Parser proto potřebuje od analyzátoru dostat další token ke zpracování včetně informací z tabulky symbolů (viz obrázek níže).

V lexikální analýze mohou nastat nejednoznačnosti, pokud je jeden symbol prefixem jiného symbolu (== apod.). Pak se hledá nejdelší symbol a je vyžadována nápověda od syntaktického analyzátoru.

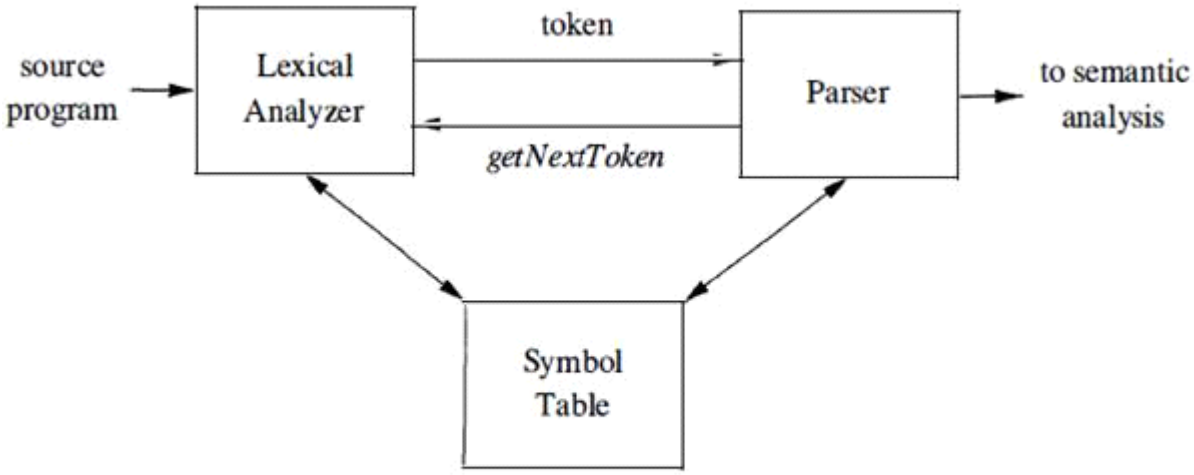


Figure 3.1: Interactions between the lexical analyzer and the parser

Často je potřeba dopředu skenovat vstup, aby se zjistilo, kde následující lexém končí. Proto lex. analyzátory typicky bufferují vstup.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Konstruktory lexikálních analyzátorů

Thursday, May 30, 2013 8:39 AM

Lex

Program LEX (Lexical Analyzer Generator) slouží k tvorbě jiných programů, které mají cosi udělat se vstupním (textovým) souborem za pomoci lexikální analýzy. Tím se myslí analýza struktur, které se dají zapsat lineárními gramatikami, konečnými automaty nebo regulárními výrazy. Typické použití LEXu je dvojí: vytvořený program pracuje samostatně, nebo slouží jako vstupní filtr pro jiný (syntaktický) analyzátor, např. bison či yacc.

Lex umožňuje vytvořit lexikální analyzátor uvedením regulárních výrazů, které popisují vzory (patterns) pro tokeny. Vstupní notace pro Lex se nazývá *Lex language* a samotný nástroj je *Lex compiler*. Kompilátor Lexu transformuje vstupní vzory do přechodového diagramu (Jádrem toho všeho je *konečný automat*.) a generuje kód do souboru lex.yy.c, ve kterém je simulován přechodový diagram.

Vstupem Lexu je soubor, obvykle s koncovkou .l, např. *lex.l*, je napsaný v jazyce Lexu a popisuje lexikální analyzátor k vygenerování (rozpoznávaná slova a akce, které se mají po jejich rozpoznání provést). Slova (tokeny) se popisují regulárními výrazy, akce v cílovém programovacím jazyku. Výstup LEXu je zdrojový kód hotového programu, který se potom musí běžným způsobem přeložit. Pokud tedy používáme variantu LEXu, která generuje výstup v jazyku C, musí být i akce zapsané v jazyku C.

Lex kompilátor překlopí lex.l do programu v C, který je vždy uložen v souboru *lex.yy.c*. Pak je tento soubor zkompilován vždy do *a.out*. Výstupem je fungující lexikální analyzátor, který bere proud vstupních znaků a vytváří z nich proud tokenů.

Hodnoty atributů (= numerický kód/pointer do tabulky symbolů/nic) jsou umístěny v globální proměnné `yylval`, kterou sdílí lexikální analyzátor a parser.

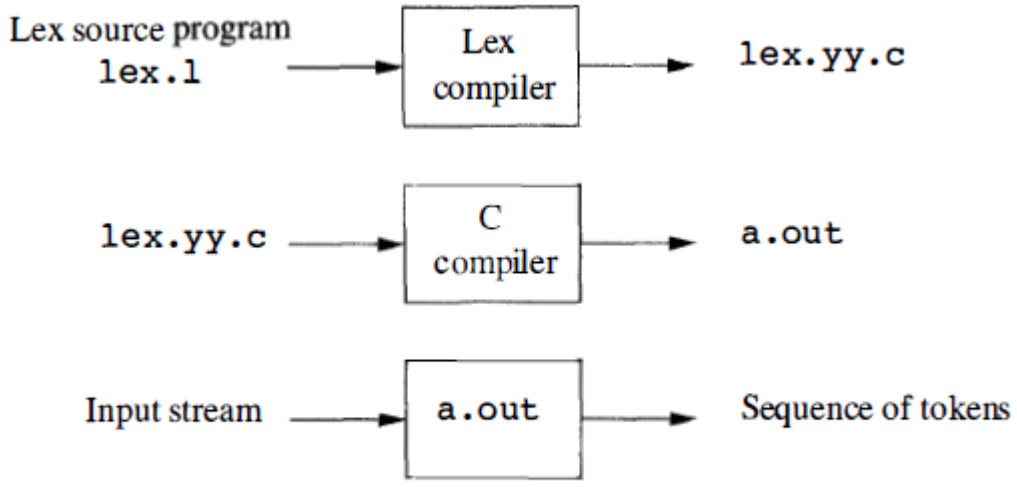


Figure 3.22: Creating a lexical analyzer with **Lex**

Struktura programu v Lexu

```
deklarace, definice
%%
popis slov a akcí
%%
další funkce (zapsané v cílovém jazyku)
```

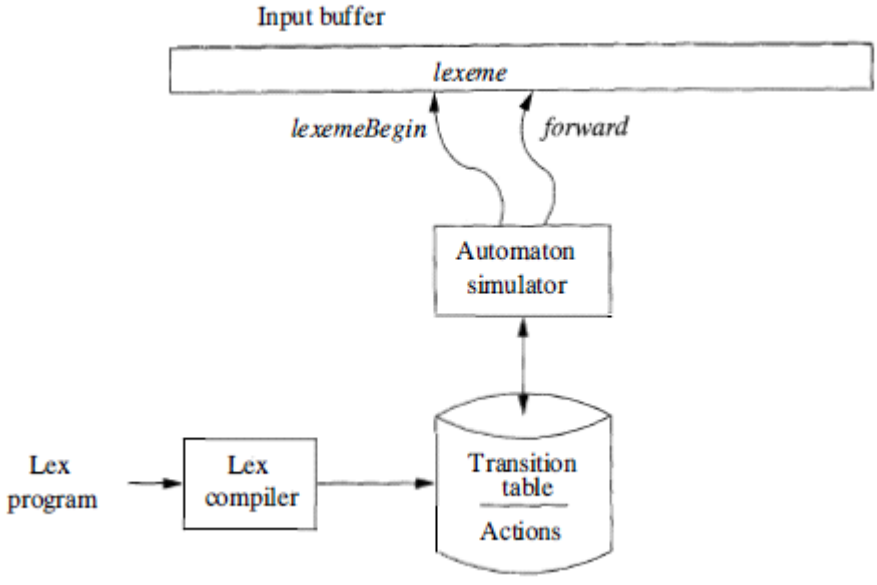
Příklad: program, který ve vstupním souboru nahradí všechny identifikátory slovem "IDENTIFIKATOR"

```
%%
[a-zA-Z_][0-9a-zA-Z_]*    printf("IDENTIFIKATOR");
%%
int main(void)
{
  yylex();
  return 0;
}
```

V popisu rozpoznávaných slov lze používat následující konstrukce:

x	znak "x"
[xy]	znak "x" nebo "y"
[x-z]	všechny znaky od "x" až k "z"
[^x]	jakýkoliv znak vyjma "x"
.	jakýkoliv znak, až na novou řádku
^x	znak "x", pokud se nachází na začátku řádky
x$	znak "x", pokud se nachází na konci řádky
x*	libovolný počet znaků "x"
x+	alespoň jeden znak "x"
x?	jeden nebo žádný znak "x"
x{m,n}	M až n výskytů znaku "x"
x|y	znak "x" nebo "y"
(x)	znak "x"
x/y	znak "x", je-li následován znakem "y"
{DEF}	doplnění definice z úvodní sekce
<y>x	znak "x", je-li splněna podmínka y

Architektura lexikálního analyzátoru generovaného Lexem



Lex z definovaných regulárních výrazů ze vstupního souboru → NKA → DKA; pravidlo: v případě konfliktů přiřazuje lexém vzoru dle nejdelšího prefixu.

Další varianty

- Flex – volně dostupná implementace Lexu, pro C.
- JLex – volně dostupná implementace Lexu, pro Javu.
- C# LEX - varianta JLex pro C#.

- PLY - implementace Lexu v Pythonu

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Bezkontextové gramatiky a zásobníkové automaty, formální popis, ekvivalence

Thursday, May 30, 2013 8:39 AM

https://class.coursera.org/compilers/lecture/38

Bezkontextové gramatiky

Bezkontextové gramatiky (BKG) jsou gramatiky typu 2 podle Chomského hierarchie.

Gramatika je bezkontextová, **tvoří-li levou stranu všech přepisovacích pravidel právě jeden neterminální symbol**

Skládají se tedy z pravidel

$$A \to \gamma$$

kde A je právě jeden neterminál a

$$\gamma$$

je řetězec terminálů a neterminálů. Pravidlo

$$S \to e$$

je povoleno, pokud se S nevyskytuje na pravé straně žádného pravidla. Jazyky generované touto gramatikou jsou **rozpoznatelné nedeterministickým zásobníkovým automatem**.

Skládají se z **terminalů, neterminálů, počátečního symbolu a přepisovacích (produkčních) pravidel; G = (N,T,P,S)**.

- T = Terminály – jde o názvy tokenů (klíčová slova *if*, *else*, symboly „(“, „)“ atd.)
- N = Neterminály – syntaktické proměnné, pomáhají definovat jazyk generovaný gramatikou; zavádějí hierarchickou strukturu jazyka, která je klíčová pro syntaktickou analýzu
- S = Počáteční symbol – jeden z neterminálů
- P = přepisovací pravidla - ve tvaru

$$A \to \gamma, \text{kde } A \in N \text{ a } \gamma \in N \cup T$$

Příklad

Jazyk

$$L = \{0^n 1^n\}$$

pro

$$n \geq 0$$

, takovýto jazyk není rozpoznatelný konečným automatem, zásobníkovým ano („konečný automat neumí počítat“). U programovacích jazyků by to třeba znamenalo, že není možné závorkovat (vnořovat kód) do libovolné úrovně.

Pro tento jazyk by platilo:

$$N = \{S\}$$
$$T = \{0,1\}$$
$$P = \{S \to 0S1, S \to e\}$$
$$S = \{S\}$$

Zásobníkový automat

Formálně je zásobníkový automat definován jako **uspořádaná sedmice (Q,T,G,δ, q_0,z_0,F)**, kde:

- Q je konečná množina vnitřních stavů,
- T je konečná vstupní abeceda,
- G je konečná abeceda zásobníku,
- δ je tzv. přechodová funkce, popisující pravidla činnosti automatu (jeho program), je definováno jako zobrazení

  $Qx(T \cup \{e\})xG^* \text{ do } QxG^*$
- q_0 je počáteční stav,
- z_0 popisuje symboly uložené na počátku v zásobníku,
- F je množina přijímajících stavů,

  $F \subseteq Q$

  .

  Je vidět, že zásobníkový automat se v podstatě skládá z konečného automatu, který má navíc k dispozici potenciálně nekonečné množství paměti ve formě zásobníku. Obsah tohoto zásobníku ovlivňuje činnost automatu tím, že vstupuje jako jeden z parametrů do přechodové funkce.

  Zásobníkový automat se od konečného automatu liší ve dvou směrech:

1. Využívá vršek zásobníku při rozhodování jaký přechod provést.
2. Může manipulovat se zásobníkem jako součást provádění přechodu.

Popis činnosti automatu

Na počátku se automat nachází v definovaném počátečním stavu a zásobník obsahuje pouze počáteční symboly. Dále v každém kroku podle aktuálního stavu, symbolů na vrcholu zásobníku a symbolu na vstupu provede přechod, při kterém může vyjmout ze zásobníku několik symbolů, vložit místo nich jiné a na vstupu přečíst další symbol. Toto se opakuje.

Po dokončení činnosti (po přečtení celého vstupu, pokud do té doby nedojde k chybě) je rozhodnuto, jestli automat vstupní řetězec přijal. K tomu mohou sloužit dvě kritéria:

- stav, ve kterém se na konci automat nachází, patří do množiny přijímajících stavů, nebo
- zásobník je na konci prázdný.

Obě definice jsou ekvivalentní, automaty na sebe lze vzájemně převádět (u druhé možnosti je možno z definice automatu zcela vypustit množinu přijímajících stavů).

Konfigurace automatu se dá popsat uspořádanou trojicí (q, w, alfa), kde q je vnitřní stav, w dosud nezpracovaná část vstupu a alfa obsah zásobníku. Na počátku práce je automat v konfiguraci (q0, w, z0).

Příklad akceptace řetězce zásobníkovým automatem: *viz cvičení 10 (LL gramatiky) na courseware FJP*

Vztah bezkontextových gramatik a zásobníkových automatů

Zásobníkové automaty jsou ekvivalentní bezkontextovým gramatikám: pro každou bezkontextovou gramatiku existuje zásobníkový automat, který generuje (akceptuje) identický jazyk generovaný touto gramatikou a naopak.

Pro danou BKG gramatiku W=(N, T, P, S) můžeme sestrojit zásobníkový automat P takový, že L(W)=L(P). Jsou dvě varianty:

1. Konstrukce zásobníkového automatu, který je modelem **syntaktické analýzy shora dolů**:
   - Q = {q} (automat má jen jeden vnitřní stav),
   - T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
   - G = N+T, tj. v zásobníku se může vyskytnout jakýkoliv symbol rozpoznávané gramatiky,
   - δ je dáno rozkladovou tabulkou,
   - q0 = q, počáteční stav automatu je q, neboť automat jiné stavy nemá,
   - z0 = S, tj. na počátku je v zásobníku startovací symbol gramatiky

- F = {}, což se interpretuje jako "automat akceptuje vyprázdněním zásobníku".
- *LL(k) gramatiky*

2. **Analýza zdola nahoru** je obecnější a vyžaduje trochu složitější automat:
   - Q = {q, r}, stav q je "pracovní", stav r "akceptační",
   - T je shodná s množinou terminálních symbolů rozpoznávané gramatiky,
   - G je v nejjednodušším případě rovno N+T+{#}, tj. sjednocení symbolů gramatiky a speciálního symbolu "#"; deterministický automat může mít množinu G složitější
   - δ je dáno rozkladovou tabulkou,
   - q0 = q,
   - z0 = #,
   - F = {r}.
   - *gramatiky: LR, SLR, LALR*

Nedeterministický syntaktický analyzátor.

Thursday, May 30, 2013 8:40 AM

Při syntaktické analýze konstruujeme derivační strom. Podle toho, jak je konstruován derivační strom věty, rozlišujeme dvě základní metody syntaktické analýzy:

1. metoda shora dolů

- derivační strom konstruujeme od kořene k listům a zleva doprava (provádíme levou derivaci)

2. metoda zdola nahoru

- postupujeme od listů směrem ke kořeni, ale také zleva doprava (provádíme pravou derivaci)

K syntaktické analýze se využívají zásobníkové automaty (ZA), které jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

- Deterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
- Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický (pohled do zásobníku nebo dále do vstupního řetězce).

Obecný popis nedeterminismu a determinismu

Základem **nedeterminismu** je tedy vždy problém **výběr správného pravidla**, ať už analýzou shora dolů nebo zdola nahoru. Pokud se nemá analyzátor podle čeho rozhodnout, prostě **prochází prostor všech řešení** buď do šířky nebo do hloubky (backtracking) a hledá to správné řešení. Tato metoda je tedy značně **neefektivní**, protože v nejhorším případě může projít všechny možnosti a nenajít žádné správné řešení, tedy správnou množinu pravidel, jejichž expanzí/redukcí lze dosáhnout požadovaného výsledku.

V případě, že chceme analyzovat vstup **deterministicky**, musíme analyzátor **zdokonalit**. Ten musí mít přesnou informaci o tom, jaké pravidlo gramatiky v danou chvíli použít. Automat může využít informaci o **dosud provedené částečné derivaci** a také o **vstupu**, který ještě nebyl zpracován. Zásobníkovému automatu, který je abstraktním modelem syntaktického analyzátoru, tedy připravíme rozkladovou tabulku (lookup table), ve které bude určeno, jaké pravidlo má analyzátor použít podle vstupu a stavu zásobníku.

Metoda shora dolů

Postup z přednášek:

A.

Konstrukce ZA, který je modelem syntaktické analýzy metodou shora dolů.

$\mathcal{P} = (\{q\}, T, N \cup T, \delta, S, \emptyset)$, kde δ je definováno takto:

1. $\delta(q, e, A) = \{(q, \alpha) : A \to \alpha \in P\}$ pro $\forall A \in N$,
2. $\delta(q, a, a) = \{(q, e)\}$ pro $\forall a \in T$.

Operaci 1. nazýváme expanzí (nahradí na vrcholu zásobníku a tím i ve větné formě neterminální symbol některou jeho pravou stranou).

Operaci 2. nazýváme srovnáním (čteného vstupního symbolu a symbolu z vrcholu zásobníku).

Tento ZA má vrchol zásobníku vždy vlevo.

Př. Zapsat $\mathcal{P}$ pro G[E] (na tabuli)

$\mathcal{P} = (\{q\}, \{(,),+,*,a\}, \{E,T,F,(,),+,*,a\}, \delta, q, E, \emptyset)$
$\delta(q, e, E) = \{(q, E+T), (q, T)\}$
$\delta(q, e, T) = \{(q, T*F), (q, F)\}$
$\delta(q, e, F) = \{(q, (E)), (q, a)\}$
$\delta(q,a',a') = \{(q, e)\}$ pro $\forall a' \in \{(,),+,*,a\}$

Např. zpracování věty a+a
Tady je vrchol zásobníku
$(q, a+a, E) \vdash (q, a+a, E + T) \vdash (q, a+a, T + T)$ to jsou expanze
$\vdash (q, a+a, F + T) \vdash (q, a+a, a + T)$ teď provedeme 2krát srovnání
$\vdash (q, +a, + T) \vdash (q, a, T)$ a opět expandujeme
$\vdash (q, a, F) \vdash (q, a, a)$ a naposledy srovnáme $\vdash (q, e, e)$

Zásobník je po přečtení vstupního řetězce prázdný, takže řetězec byl akceptován

Derivační strom konstruujeme od kořene (ohodnoceného startovním symbolem) dolů k listům, zleva doprava podle levé derivace. Jedná se o zásobníkový automat LL. Počáteční konfigurace automatu se dá popsat uspořádanou trojicí **(q, w, alfa)**, kde:

q =vnitřní stav
w = dosud nezpracovaní část vstupu
alfa = obsah zásobníku
Na počátku práce je automat v konfiguraci **(q_0, w, z_0)**, napr. (q, abaaab, s).

Pokud jen generujeme větu v gramatice, můžeme v případě více pravidel se stejnou levou stranou náhodně vybírat. Naším úkolem však bývá spíše analýza již existující věty. Zde již náhoda nepřipadá v úvahu, protože posloupnost pravidel pro levou derivaci již nemusí být jednoznačná. Potřebujeme automat, který tuto analýzu provádí, a tento automat musí mít možnost jednoznačně vybírat mezi pravidly to správné.

Existují dva postupy analýzy (nedeterministická a deterministická):

- **analýza s návratem** (nedeterministická)
  - postupně zkoušíme vhodná pravidla. Nejdřív první, pokračujeme dále ve výpočtu, a když se ukáže, že pravidlo nevyhovuje (dostaneme se do slepé uličky), vrátíme se zpátky a vyzkoušíme druhé pravidlo atd. Tato metoda je sice účinná. ale zbytečně pomalá.
  - Rekurzivní sestup.

- **deterministická analýza**
  - při výběru pravidla se řídíme dalšími informacemi. Může to být *pohled do budoucnosti*, kdy se díváme dále do vstupní posloupnosti symbolů a řídíme se tím, co později dostaneme na vstupu. Nebo například kontrolujeme obsah zásobníku (nestačí nám pouze vidět ten symbol, který ze zásobníku vyjímáme, ale i další, které jsou pod ním).
  - Prediktivní LL parser

Metoda zdola nahoru

Postup z přednášek:

Konstrukce ZA, který je modelem syntaktické analýzy metodou zdola nahoru.

$P = (\{q, r\}, T, N \cup T \cup \{\#\}, \delta, q, \#, \{r\})$, kde δ je definováno takto:

1. $\delta(q, a, e) = \{ (q, a) \}$ pro $\forall a \in T$,
2. $\delta(q, e, \alpha) = \{ (q, A) : A \to \alpha \in P \}$,
3. $\delta(q, e, \#S) = \{ (r, e) \}$.

Operaci 1. nazýváme přesun (přesun vstupního symbolu na vrchol zásobníku).

Operaci 2.nazýváme redukce (náhrada pravé strany pravidla na vrcholu zásobníku a tím i ve větné formě stranou levou).

Operace 3. je přijetí.

Tento ZA má vrchol zásobníku vpravo.
Konfiguraci budeme zapisovat ve tvaru: (stav, zásobník, vstup).
Zřetězením stavu zásobníku se zbytkem vstupu pak uvidíme jednotlivé větné formy

Př. Zapsat $\mathcal{P}$ pro G[E] (na tabuli)

- $\mathcal{P} = (\{q,r\}, \{(,),+,*,a\}, \{\#,E,T,F,(,),+,*,a\}, \delta, q, \#, r)$

  $\delta(q, a', e) = \{ (q, a') \}$ pro $\forall a \in T$,
  $\delta(q, e, E+T) = \{ (q, E) \}$
  $\delta(q, e, T) = \{ (q, E) \}$
  $\delta(q, e, T*F) = \{ (q, T) \}$
  …
  $\delta(q, e, \#E) = \{ (r, e) \}$

Např. zpracování věty a+a

Vrchol

$(q, \#, a+a) \vdash (q, \#a, +a) \vdash (q, \#F, +a) \vdash (q, \#T, +a)$
$\vdash (q, \#E, +a) \vdash (q, \#E+, a) \vdash (q, \#E+a, e) \vdash (q, \#E+F, e)$
$\vdash (q, \#E+T, e) \vdash (q, \#E, e) \vdash (r, e, e)$

ZA konstruované dle A. i B. jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

a) Deterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
b) Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický.

Pozn.: Obsah zásobníku zřetězený se zbytkem vstupu je větnou formou.

- Konstruujeme derivační strom zdola od listů nahoru ke kořeni, přičemž postupujeme zleva doprava.

- Stejně jako u první metody i zde budeme používat lineární rozklad, tentokrát pro pravou derivaci - je to proces nalezení pravého rozkladu věty (LR gramatika).
- I zde musíme rozhodovat, která pravidla chceme použít. Tentokrát však nejde o pravidla se stejnou levou stranou (pro stejný neterminál), ale rozhodujeme se mezi pravidly, která mají podobnou pravou stranu a jsou proto použitelná pro tentýž podřetězec větné formy.

Řešíme to podobně jako u předchozí metody:

- **analýza s návratem** (nedeterministicky)
  - Vybereme ve větné formě jeden podřetězec (jako první vybíráme ten, který začíná nejvíc nalevo, je co nejdelší a je shodný s pravou stranou některého pravidla), přepíšeme neterminálem na pravé straně pravidla a pokračujeme v konstrukci derivačního stromu.
  - Pokud zjistíme, že tento krok nevede k úspěchu, vyzkoušíme jiný podřetězec atd (backtracking). Tato metoda je příliš časově náročná.
- **deterministická analýza (LALR, SLR)**
  - Využíváme další informace získané při překladu, např. obsah nepřečtené části vstupního kódu nebo obsah zásobníku.

ZA konstruované výše uvedenými metodami jsou obecně nedeterministické (nepoužitelné pro SA). Pro konstrukci SA lze použít buď:

a) Neterministickou simulaci nedeterministického ZA = algoritmus syntaktické analýzy s návraty.
b) Zdokonalit konstrukci ZA tak, aby byl pro určitou třídu BKG deterministický.

Pozn.: Obsah zásobníku zřetězený se zbytkem vstupu je větnou formou.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Derivace a derivační strom, víceznačnost gramatiky

Thursday, May 30, 2013 8:40 AM

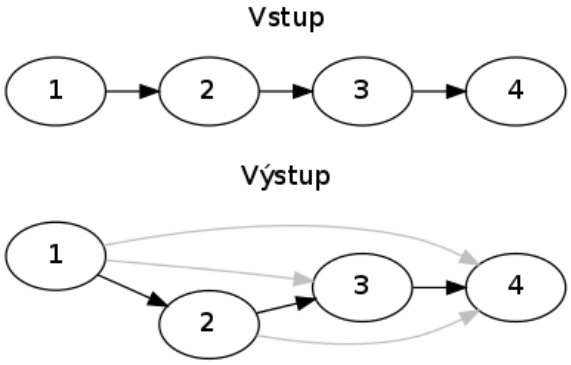
Derivace

- Posloupnost kroků odvození terminálu pomocí přepisovacích pravidel gramatiky
- Derivační pohled odpovídá konstrukci parsovacího (syntaktického) stromu shora dolů (top-down).
- Parsování zdola nahoru (bottom-up) je spjato s pravými derivacemi.
- Podle toho, který neterminál nahradit v každém kroku derivace, se rozlišují **levá a pravá derivace**.

DERIVACE řetězce α je posloupnost kroků odvození α pomocí přepisovacích pravidel gramatiky

$S = \alpha_1 => \alpha_2 => \ldots => \alpha_n = \alpha$

Dtto $S =>^* \alpha$ pozn.: $=>^*$ je uzávěr relace => (všechny přechody kam se dá transitivně dostat)



PŘÍMÁ DERIVACE: α A β => α γ β ,kde A -> γ ϵ P (pozn.: P je množina pravidel, pomocí kterých lze odvodit jazyk.)

Derivační strom

Derivační strom (parse tree) je orientovaný acyklický graf a je grafickou reprezentací, která říká, v jakém pořadí byla přepisovací pravidla uplatňována na neterminály, tedy jak vznikla věta jazyka.

- Kořen stromu je označen startovacím symbolem gramatiky
- Každý vnitřní uzel je ohodnocen neterminálními symboly.
- Listy jsou ohodnoceny terminálními symboly.
- Listy se čtou zleva doprava a dávají větu (generovanou gramatikou).
- Jestliže uzly $n_1, n_2, \ldots, n_k$ jsou bezprostřední následníci uzlu n, jsou ohodnoceny symboly $A_1, A_2, \ldots, A_k$ a uzel *n* je ohodnocen A, pak v množině pravidel gramatiky existuje pravidlo $A \rightarrow A_1A_2 \ldots A_k$.
- Není třeba značit orientaci hran.

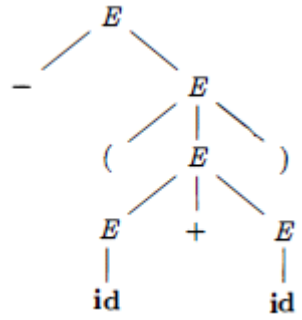
Jiná definice jazyka generovaného gramatikou je množina vět, které mohou být vytvořeny derivačním stromem. Proces hledání derivačního stromu pro danou větu (řetězec terminálů) se nazývá **parsování** tohoto řetězce

Derivační strom ignoruje variace v pořadí, v jakém jsou symboly přepisovány. Proto je mezi derivacemi a derivačními stromy vztah 1:N – např.:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Jsou různé derivace, jejich derivační strom ale vypadá stejně:



Pro získání jednoznačného derivačního stromu pro derivaci se proto užívá buď pravá a nebo levá derivace.

Víceznačnost gramatik

Gramatika, která generuje větu, pro níž lze sestavit aspoň dva různé derivační stromy, je víceznačná. Jinak řečeno: je to taková gramatika, která produkuje více než jednu levou nebo víc než jednu pravou derivaci pro tutéž větu.

Příklad: Pro gramatiku

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

umožňuje vytvořit dvě levé derivace pro id + id * id (násobení není upřednostněno před sčítáním):

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

- **Nutnou podmínkou jednoznačnosti gramatiky je, aby pro žádný neterminální symbol neexistovalo jak pravidlo rekurzivní zprava, tak i pravidlo rekurzivní zleva**
- **Problém nejednoznačnosti bezkontextových jazyků je algoritmicky nerozhodnutelný.**

Je potřeba buďto vytvořit jednoznačné gramatiky pro kompilaci aplikací, nebo u nejednoznačných gramatik zavést dodatečná pravidla, která řeší případné nejednoznačnosti.

Odstranění levé rekurze

- **Levorekurzivní gramatiku nelze použít k analýze shora dolů**

Odstranění pravidla rekurzivního zleva:

- Nechť je dána BKG G = (N, T, P, S), ve které,
  - $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
    jsou všechna A pravidla v P a žádné z β nezačíná A.

- Pak G' = (N ∪ { A'}, T, P', S), kde P' obsahuje místo uvedených pravidel pravidla:
  - $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \mid \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_m \mid \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A'$

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Deterministická syntaktická analýza

Thursday, May 30, 2013 8:41 AM

Např. s pomocí **prediktivního parseru** = rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Deterministická syntaktická analýza využívá další informace získané při překladu – obsah zásobníku a obsah nepřečtených vstupních tokenů. Na základě těchto informací umožňují následující funkce vhodný výběr přepisovacích pravidel, a tím prediktivní parsování:

FIRST(A) = množina terminálů, kterými mohou začínat řetězce odvozené z A **(terminály na začátku A)**

Funkce first zjišťuje, co vznikne přepsáním jednotlivých neterminálů na levé straně všech pravidel.

- A => bxayB → b náleží FIRST(A)

Algoritmus výpočtu

- FIRST(A) pro A = terminál nebo *e:* je terminál/*e*
- Když je A neterminál:
  - Je to první terminál ve všech přepisovacích pravidlech s A na levé straně
  - Pokud jsou v přepisovacím pravidle na P straně jen neterminály, hledá se FIRST prvního neterminálu na pravé straně
  - Pokud lze nějaký z těchto neterminálů přepsat na prázdný řetězec *e*, je potřeba se podívat na *first* neterminálu následujícího po něm v některém z přepisovacích pravidel

FOLLOW(A) = množina terminálů, které mohou následovat za A v některé větné formě v derivacích **(terminály hned za A)**

- S => bxCAyZ → y náleží FOLLOW(A)

Algoritmus výpočtu

1. Polož FOLLOW (A) = θ

2. Je-li A počáteční symbol G, přidej e do FOLLOW(A)

3. Pro všechny pravé strany pravidel z G tvaru α A β přidej FIRST (β) do FOLLOW (A), nepřidávej ale e.

4. Je-li v G pravidlo L → α A nebo L → α A β , kde FIRST (β)

obsahuje e, pak přidej do FOLLOW (A) množinu FOLLOW (L)

Vytváříme vždy pro všechny neterminály zároveň!

$FIRST_k(A)$, $FOLLOW_k(A)$ = zobecnění na množiny terminálních řetězců o délce nejvýše *k*

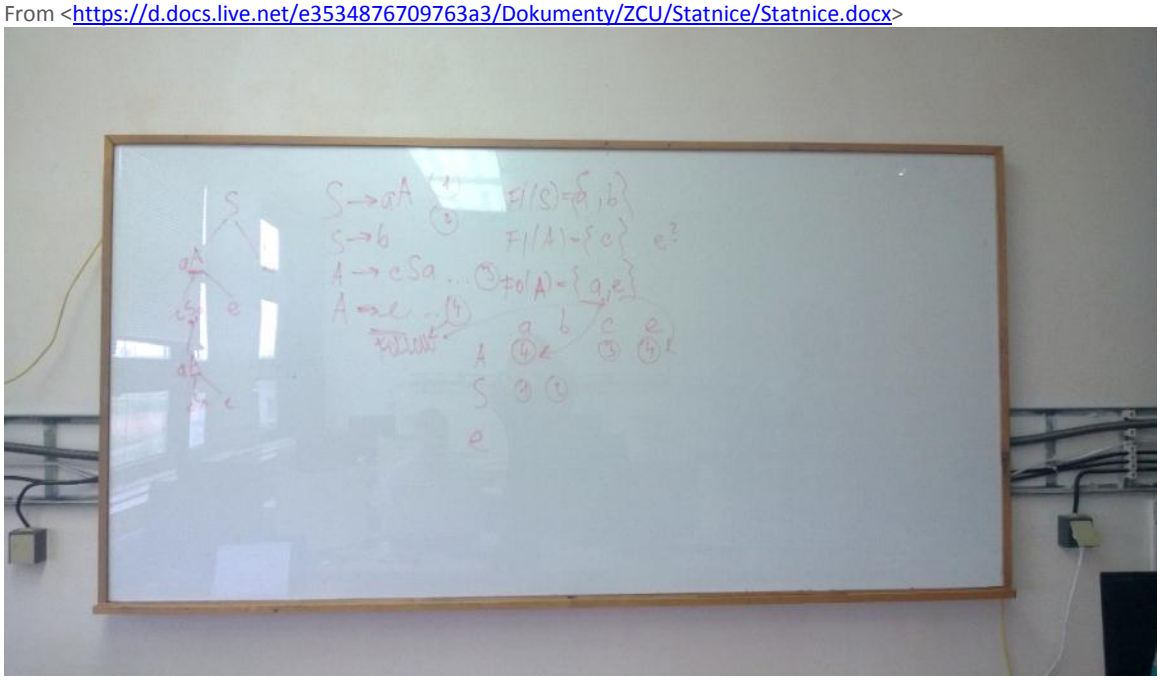
Tyto funkce slouží k vytvoření **rozkladové tabulky**, která nahrazuje přechodovou funkci. V první řádce jsou uvedeny všechny možné vstupy, v prvním sloupci všechny možné stavy vrcholu zásobníku

(vč. dna zásobníku #). Má stavy:

- **Srovnání (pop)** – na vstupu i na vrcholu zásobníku jsou stejné hodnoty
- **Přijetí (accept)** – bylo dosaženo dna zásobníku a přijímá se prázdný symbol *e*
- **expanze (expand)** – aplikace přepisovacího pravidla, které je uvedené v buňce tabulky určené vstupem (který sloupec) a vrcholem zásobníku (který řádek)
- **chyba (error)** – pokud pro vstup a hodnotu na vrcholu zásobníku je buňka v tabulce prázdná → vstupní řetězec není větou jazyka

Syntactic Analysis in terms of **bottom up algorithms: LR, SLR, LALR**

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>



Rekurzivní sestup

Thursday, May 30, 2013 8:41 AM

https://class.coursera.org/compilers/lecture/25

Top Down

Shora

Zleva doprava

Rekurzivní sestup nebo také **rekurzivní sestupný parser** postupuje shora dolů a je sestaven ze vzájemně se volajících procedur. Každá taková procedura obvykle implementuje jedno přepisovací pravidlo gramatiky. (Kromě něj do top-down parserů patří prediktivní parsery založené na LL(k) gramatikách.)

Prediktivní parser je rekurzivně sestupný parser, který nevyžaduje zpětné kroky. Je ho možné vytvořit jen pro LL(k) gramatiky, což jsou bezkontextové gramatiky, pro které existuje kladné *k*, které umožní rekurzivně sestupnému parseru rozhodnout se, které přepisovací pravidlo použít na základě *k* dalších načtených symbolů. LL(k) gramatiky vylučují mnohoznačnost a levou rekurzi. Jakákoliv bezkontextová gramatika může být transformována na ekvivalentní nelevorekurzivní gramatiku, ale odstranění levé rekurze ne vždy vede k LL(k) gramatice. Prediktivní parser běží v lineárním čase.

Rekurzivní sestup s návratem je technika určování použitého produkčního pravidla zkoušením všech pravidel. Není limitován na LL(k) gramatiky, ale nemá zaručeno skončit, pokud gramatika není LL(k). Může vyžadovat exponenciálni čas pro svůj běh.

Princip

Hlavní myšlenka je taková, že pro každý neterminál gramatiky je implementována příslušná funkce v programu.

- každému neterminálnímu symbolu *A* odpovídá procedura *A*
- tělo procedur je dáno pravými stranami pravidel pro *A*
- pravé strany musí být rozlišitelné na základě symbolů vstupního řetězce
- je-li rozpoznána pravá strana, pak v případě neterminálního symbolu vyvolá *A* proceduru pro rozpoznaný neterminální symbol, v případě terminálního symbolu, ověří *A* jeho přítomnost ve vstupním řetězci a zajistí přečtení dalšího znaku ze vstupu
- rozpoznané pravidlo analyzátor oznámí (např. jeho číslo)
- chybnou strukturu vstupního řetězce oznámí chybovým hlášením

Terminál na pravé straně je porovnán s dalším vstupním symbolem. Pokud se shodují, přejde se na další vstupní symbol a na další symbol na pravé straně. V opačném případě je nahlášena chyba.

O neterminál na pravé straně je postaráno voláním příslušné funkce. Po jejím vykonání se pokračuje dalším symbolem na pravé straně.

Pokud na pravé straně už nejsou žádné symboly, funkce končí (function returns).

Takto se postupně volají všechny funkce, až se nakonec opět ocitneme ve funkci pro startovací symbol, která byla zavolána jako první. Ta také oznamuje úspěšný průběh, pokud se prošel celý vstupní řetězec.

```
    void A() {
1)      Choose an A-production, A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Pseudokód výše je nedeterministický, protože začíná volbou A-přepisovacího pravidla, které se používá blíže nepopsaným způsobem.

Příklad: http://info.lu2.name/soubory/prekl_06_604.pdf

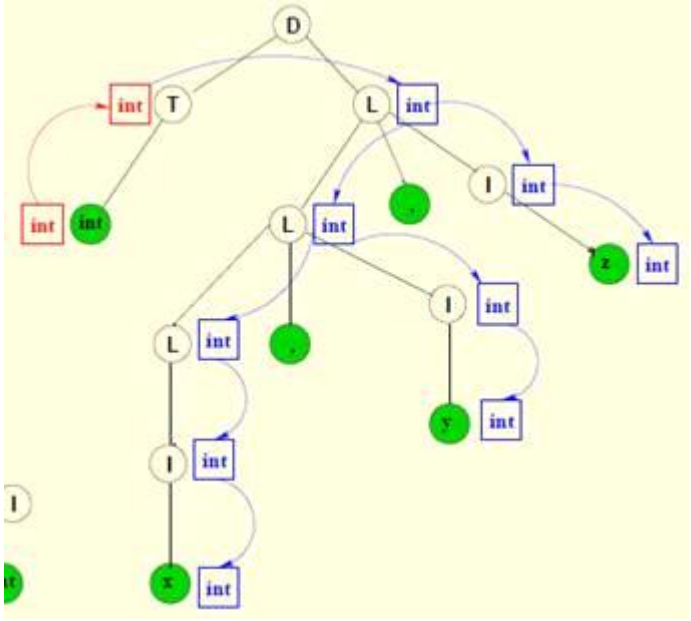
Obecně může rekurzivní sestup potřebovat backtracking, tzn. někdy je třeba se vrátit a opakovaně číst vstup. Backtracking je však potřeba zřídkakdy (dá se eliminovat volbou vhodné gramatiky - tedy zavedením jistých omezení). Kód výše neumožňuje backtracking, bylo by je nutno modifikovat – v řádce 7 se pak vrátit na řádku 1 a zvolit jiné pravidlo, popř. nahlásit chybu, když už žádné další nejde použít.

Sémantické zpracování

Při rekurzivním sestupu se může provádět také sémantické zpracování. Sémantické zpracování zahrnuje vyhodnocení atributů symbolů v derivačním stromu. Atributy = vlastnosti gramatických symbolů nesoucí sémantickou informaci (hodnota, adresa, typ, scope, spojitost mezi formálními a skutečnými parametry apod.).

Způsoby vyhodnocení:

1. procházením stromem od listů ke kořenu = **syntetizované atributy**
2. procházením stromem od rodiče k potomkovi, od staršího bratra k mladšímu = dědičné atributy (např vícenásobné deklarace v C – int x,y,z)



Je nutné doplnit procedury lex. analýzy (LA) i syntakt. analýzy (SA) takto:

- LA bude předávat s přečteným vstupním symbolem i jeho atributy.
- procedury SA pro neterminály doplnit o:
  - vstupní parametry odpovídající dědičným atributům
  - výstupní parametry odpovídající syntetizovaným atributům
  - zavést lokální proměnné pro uložení atributů pravostranných symbolů
  - před vyvoláním procedury korespondujícího neterminálu z pravé strany vypočítat hodnoty

jeho dědičných atributů
    - na konec procedury popisující pravou stranu pravidla zařadit příkazy vyhodnocující syntetizované atributy

Vlastnosti

- pro metodu rekurzivního sestupu, tj. analýza shora dolů, se používají *LL* gramatiky (levorekurzivní se nedají použít, protože by se program mohl zacyklit voláním pořád té samé procedury)
- jednoduchá *LL* gramatika je taková gramatika, kde levou stranu tvoří právě jeden neterminální symbol a kde každá pravá strana začíná terminálním symbolem
- navíc musí platit, že např. pro pravidla *A->...* jsou počáteční symboly různé
- obecná *LL* gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Principy a podmínky LL analýzy

Thursday, May 30, 2013 8:42 AM

LL-gramatika

LL gramatika je jakákoliv gramatika, z níž se dá udělat rozkladová tabulka pro LL parser. **LL(k) parser** se kouká při parsování věty na následujících *k* tokenů, aby věděl, co dál. Pokud takový parser může být použit pro nějakou gramatiku, aniž by se musel použít backtracking, jedná se o **LL(k) gramatiku**.

Aby se ze vstupní gramatiky dala udělat LL(1) gramatika – eliminace levé rekurze, levá faktorizace (eliminace překrývajících se množin FIRST, tj. "rozsekání" pravidel se stejným začátkem pravé strany na několik menších, už jednoznačných)

př: STAT => if EXP then STAT | if EXP then STAT else → STAT => if EXP then STAT ElsePart; ElsePart => else STAT | e)

Podmínky

- Nesmí být přítomna levá rekurze.
- Nesmí dojít k first-follow (u neterminálu, který se přepisuje na "e") kolizi, first-first kolizi

Třídy jazyků LL(k)

L = Left to right -> vstupní text (soubor) čteme zleva doprava

L = Left parse -> vytváříme levý rozklad

K = při rozhodování mezi pravidly potřebujeme vidět nejvýše *k* znaků z nepřečtené části vstupu

Tzn.: **LL(k)** gramatika provádí deterministický rozbor *čtením textu z **L**eva doprava*, s *použitím **L**evé derivace* a *prohlédnutí **k** dalších symbolů vstupního textu*.

- gramatika je typu **LL(k)**, jestliže ji lze použít pro deterministickou syntaktickou analýzu metodou shora dolů (tj. vytváříme levý rozklad) a při rozhodování mezi pravidly potřebujeme znát nejvýše **k** symbolů ze vstupu.
- jazyk je typu **LL(k)**, pokud je generován některou **LL(k)** gramatikou

LL(0) gramatika

- lze určit správné pravidlo aniž bychom předem potřebovali vidět nějaký znak na vstupu
- každý neterminál musí mít jen jednu jedinou pravou stranu (jen jedno přepisovací pravidlo)
- neumožňuje rekurzi
- prostě jen určují, jestli sekvence patří do jazyka nebo ne, žádné rozhodování není potřeba.
- LL(0) gramatiky jsou nevhodné pro popis programovacích jazyků, protože zde není možná rekurze a pro každý neterminál existuje právě jedno pravidlo (důsledek faktu, že gramatika se nemůže rozhodnout podle následujícího vstupního symbolu), tudíž mohou generovat jen jazyk s jediným slovem.
  From <http://cs.wiki pedia.org/wiki/LL_syntaktick%C3%BD_analyz%C3%A1tor>

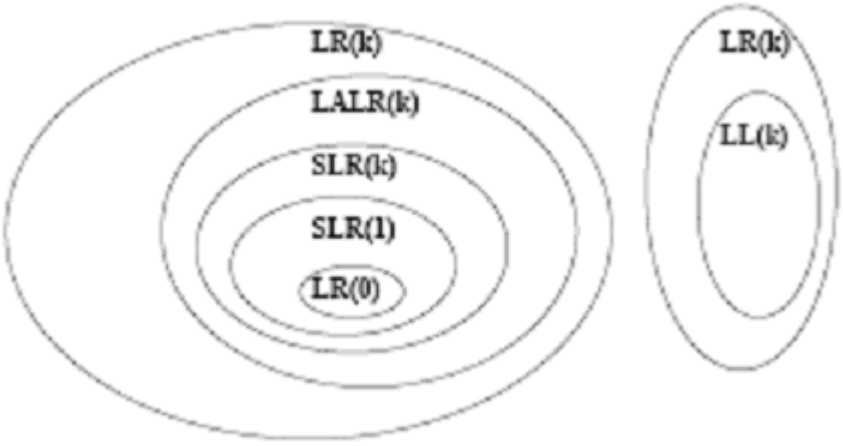
- Příklad:

```
G == id name lastname
id == [0-9]+
name == string
lastname == string
string == <unicode>+
```

LL(1) gramatika

- pro danou gramatiku *G* se vystačí při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky 1 -> proto LL(1) je **gramatika silná**
- **jednoduchá LL (1) gramatika** je taková bezkontextová gramatika jestliže platí:
  - pravá strana každého pravidla začíná terminálním symbolem např. A -> aB
  - pokud mají 2 pravidla stejnou levou stranu, pak pravé strany začínají různými terminálními symboly např. A -> aB, A->bB
  - to znamená, že v každém políčku rozkladové tabulky bude právě jeden element
- **Obecná LL(1):**
  - gramatika nemá omezení, ale musí pro ni existovat rozkladová tabulka

Mohutnosti gramatik



Typy analýzy

- shora (top-down)
- sdola (bottom-up) (vyžadují LR gramatiku, takže se netýkají této otázky)

Analýza shora = analýza top-down

- Při hledání derivace začínáme počátečním symbolem a snažíme se dostat k hledanému slovu
- LL analýza: hledáme levou derivaci, vstupní slovo analyzujeme zleva
  - Přesně určuje volbu pravidel při analýze a umožňuje jednoznačný postup při odvození
  - Gramatika, která je jednoznačná a lze ji takto analyzovat: LL gramatika
  - Využívá se zásobníkový automat
- LL(k) označení gramatiky pro LL analýzu, číslo k určuje, kolik následujících symbolů na vstupu je nutné znát pro analýzu slova
- LL(1): nejpoužívanější gramatika, stačí znát jeden následující symbol
  - Jde vlastně o variantu rekurzivního sestupu bez backtrackingu
- LL(0): umožňuje jen jazyky s konečným počtem slov
- LL gramatiky s k>1 lze převést na LL gramatiky s k = 1
  - Existují přesné popisy, jak jednotlivá pravidla nahrazovat (přidávají se neterminály a pravidla se upravují, aby při analýze stačilo znát jeden další symbol)

LL parsery = parsery s analýzou top-down

LL parsery používají parsing **shora dolů**, zpracovávají vstup zleva doprava a konstruují nejlevější derivaci. Proto se také nazývá L (left-to-right) L(leftmost derivation). Občas se setkáváme s označením LL(k), kde k značí počet tokenů, které potřebujeme znát při rozhodování o průběhu další analýzy bez toho, aby bylo třeba používat backtracking (= prediktivní parser). Také se v této souvislosti používá pojem look-ahead. Prakticky do nedávné doby se tyto gramatiky příliš nepoužívaly, ovšem na počátku 90. let minulého století došlo ke změně přístupu.

Syntaktická analýza LL gramatik

Budeme se zabývat algoritmem syntaktické analýzy, který vytváří derivační strom analyzovaného řetězce směrem shora dolů. Základní princip syntaktické analýzy můžeme v tomto případě formulovat takto:

Je dána bezkontextová gramatika G= (N,T,P,S) a řetězec w = $a_1\ a_2 \ldots a_n$, který je větou z L(G). Pak existuje levá derivace

$S = \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_n = w.$

Vzhledem k tomu, že derivace je levá, má každá větná forma γi tvar:

$\gamma i = a_1\ a_2 \ldots a_j\ A_i\ \beta_i,$

kde a_1, a_2 …, a_j jsou terminální symboly, A_i je neterminální symbol, β_i je řetězec terminálních a neterminálních symbolů. Přitom řetězec $a_1a_2 \ldots a_j$ je předponou věty w, $j \geq 0$.

Podmínky LL analýzy

Předpokládejme, že $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ jsou všechna pravidla v P s neterminálním symbolem A na levé straně. Pak základní problém syntaktické analýzy metodou shora dolů spočívá v nalezení toho pravidla $A \rightarrow \alpha_k$, jehož aplikací dostaneme z větné formy γ_i větnou formu $\gamma_i + 1$.

Pro výběr pravidla $A \rightarrow \alpha_k$, je možno použít:

1. informaci o dosavadním průběhu (historii) analýzy,
2. informaci o dosud nepřečtené části vstupního řetězce (dopředu prohlíženém řetězci omezené délky).

Pokud tyto informace vždy stačí k jednoznačnému výběru pravidla $A \rightarrow \alpha_k$, pak se gramatika G nazývá *LL gramatika*. Název je odvozen od toho, že při čtení vstupního řetězce zleva je vytvářen levý rozklad. Při syntaktické analýze LL gramatik jsou do zásobníku ukládány řetězce, které odpovídají levým větným formám nebo takovým jejich příponám, které vzniknou odejmutím předpony tvořené řetězcem terminálních symbolů.

Základními operacemi syntaktického analyzátoru pro LL gramatiky (LL analyzátoru) jsou:

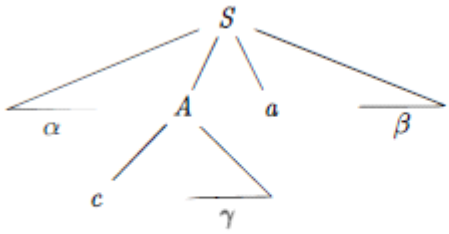
- **Expanze** – neterminální symbol na vrcholu zásobníku je nahrazen pravou stranou vybraného pravidla
- **Srovnání** – terminální symbol na vrcholu zásobníku se ze zásobníku vyloučí, jestliže je shodný se symbolem, který byl ze vstupního řetězce přečten.
- **Přijetí** – vstupní řetězec je přečten a zásobník je prázdný.
- **Chyba** – ve všech ostatních případech.

Pokud pro danou gramatiku G vystačíme při rozhodování o výběru pravidla pro expanzi s informací o dopředu prohlíženém řetězci délky nejvýše *k*, pak se gramatika G nazývá *silná LL(k) gramatika*. Při analýze silných LL(k) gramatik jsou do zásobníku ukládány přímo symboly gramatiky a syntaktický analyzátor je řízen rozkladovou tabulkou.

Funkce FIRST a FOLLOW

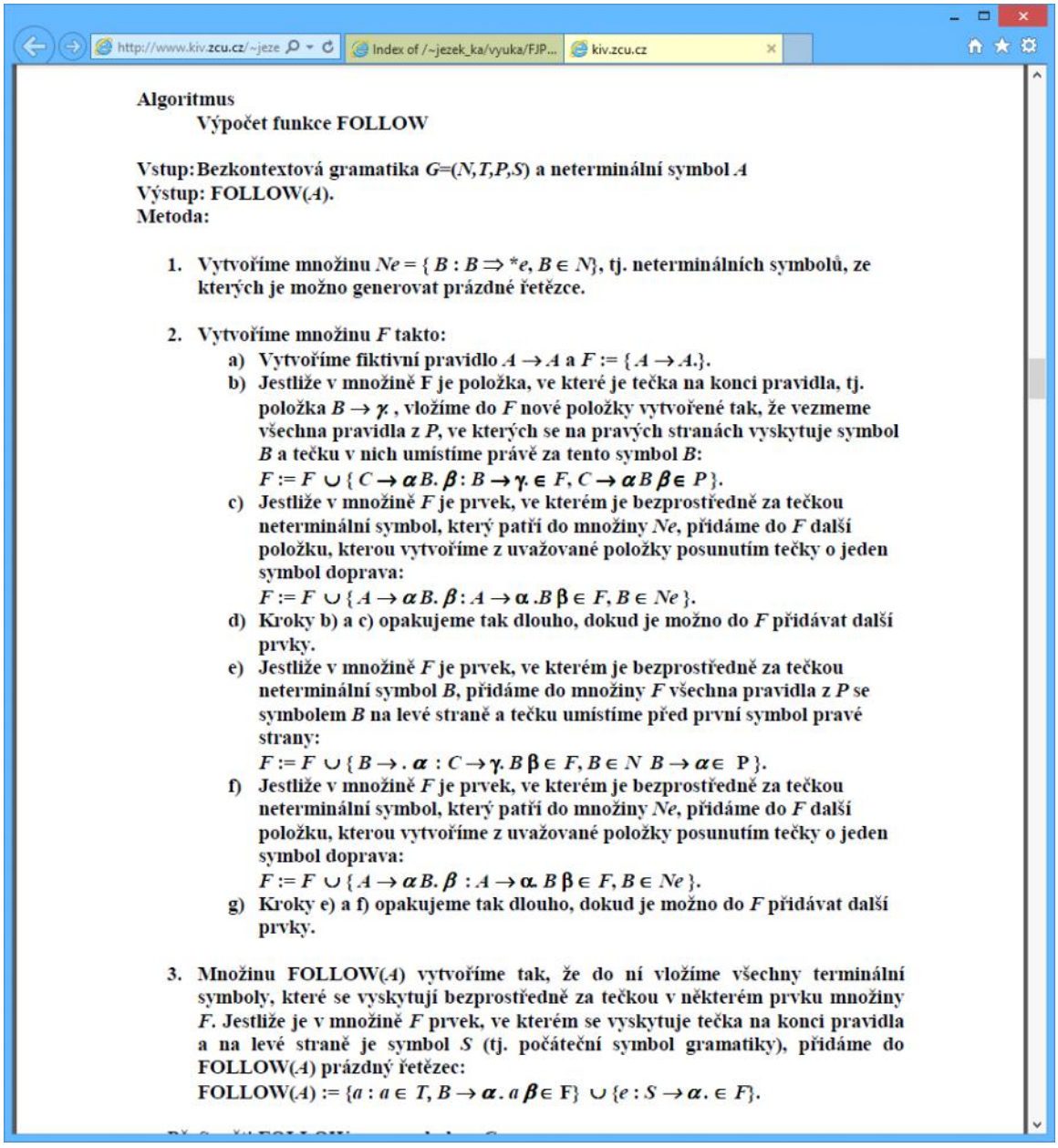
Konstrukce jak top-down, tak bottom-up parserů používá dvě funkce, FIRST a FOLLOW, spojené

s gramatikou G. Při parsování shora dolů nám FIRST a FOLLOW říkají, které přepisovací pravidlo uplatnit v závislosti na dalším vstupním symbolu. Během zotavení z chyby při panic módu mohou být množiny tokenů získané pomocí FOLLOW použity jako synchronizační tokeny.



Terminal c is in FIRST(A) and a is in FOLLOW(A)

Popis LL gramatik a LL analyzátoru



Algoritmus
Výpočet funkce FOLLOW

Vstup: Bezkontextová gramatika $G=(N,T,P,S)$ a neterminální symbol A
Výstup: FOLLOW(A).
Metoda:

1. Vytvoříme množinu $Ne = \{ B : B \Rightarrow^* e, B \in N\}$, tj. neterminálních symbolů, ze kterých je možno generovat prázdné řetězce.

2. Vytvoříme množinu F takto:
   a) Vytvoříme fiktivní pravidlo $A \rightarrow A$ a $F := \{ A \rightarrow A.\}$.
   b) Jestliže v množině F je položka, ve které je tečka na konci pravidla, tj. položka $B \rightarrow \gamma .$, vložíme do F nové položky vytvořené tak, že vezmeme všechna pravidla z P, ve kterých se na pravých stranách vyskytuje symbol B a tečku v nich umístíme právě za tento symbol B:
   $F := F \cup \{ C \rightarrow \alpha B. \beta : B \rightarrow \gamma. \in F, C \rightarrow \alpha B \beta \in P \}$.
   c) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol, který patří do množiny Ne, přidáme do F další položku, kterou vytvoříme z uvažované položky posunutím tečky o jeden symbol doprava:
   $F := F \cup \{ A \rightarrow \alpha B. \beta : A \rightarrow \alpha .B \beta \in F, B \in Ne \}$.
   d) Kroky b) a c) opakujeme tak dlouho, dokud je možno do F přidávat další prvky.
   e) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol B, přidáme do množiny F všechna pravidla z P se symbolem B na levé straně a tečku umístíme před první symbol pravé strany:
   $F := F \cup \{ B \rightarrow . \alpha : C \rightarrow \gamma. B \beta \in F, B \in N \; B \rightarrow \alpha \in P \}$.
   f) Jestliže v množině F je prvek, ve kterém je bezprostředně za tečkou neterminální symbol, který patří do množiny Ne, přidáme do F další položku, kterou vytvoříme z uvažované položky posunutím tečky o jeden symbol doprava:
   $F := F \cup \{ A \rightarrow \alpha B. \beta : A \rightarrow \alpha. B \beta \in F, B \in Ne \}$.
   g) Kroky e) a f) opakujeme tak dlouho, dokud je možno do F přidávat další prvky.

3. Množinu FOLLOW(A) vytvoříme tak, že do ní vložíme všechny terminální symboly, které se vyskytují bezprostředně za tečkou v některém prvku množiny F. Jestliže je v množině F prvek, ve kterém se vyskytuje tečka na konci pravidla a na levé straně je symbol S (tj. počáteční symbol gramatiky), přidáme do FOLLOW(A) prázdný řetězec:
   FOLLOW(A) := $\{a : a \in T, B \rightarrow \alpha . a \beta \in F\} \cup \{e : S \rightarrow \alpha . \in F\}$.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Řešení kolizí: http://www.kiv.zcu.cz/~lobaz/fjp/fjp11.html

Vnitřní jazyky překladačů – druhy, použití v jednotlivých fázích překladu, překlad jednoduchých jazykových konstrukcí

Thursday, May 30, 2013 8:42 AM

Po ukončení syntaktické a sémantické analýzy generují některé překladače explicitní intermediální reprezentaci zdrojového programu (mezikód). Intermediální reprezentaci můžeme považovat za program pro nějaký abstraktní počítač. Tato reprezentace by měla mít dvě důležité vlastnosti: měla by být jednoduchá pro vytváření a jednoduchá pro překlad do tvaru cílového programu. Intermediální kód slouží obvykle jako podklad pro optimalizaci a generování cílového kódu. Může však být také konečným produktem překladu v interpretačním překladači, který vygenerovaný mezikód přímo provádí. Intermediální reprezentace mohou mít různé formy.

Postfixová notace

operátory následují ihned za operandy

- A B C * D + - => A – (B * C + D)
- efektivní zpracování pomocí zásobníku, musíme vědět prioritu operátorů

Instrukce postfixoveho zapisu napr. Lit 0, A = uloz konstantu A do zasobniku, opr 0, A = proved instrukci A…

Prefixová notace

operátory a pak operandy

- + A B + C D => (A + B) * (C + D)

Tříadresový kód

Abstraktní forma mezikódu sestávající ze sekvence příkazů ve tvaru **x := y op z**, kde x, y a z jsou jména, konstanty nebo dočasné proměnné, op je nějaký operátor. Na levé straně je **adresa**, na pravé **instrukce.** Adresou může být název (ze zdrojového programu, je pak nahrazen pointerem do jeho tabulky symbolů), konstanta, kompilátorem generovaná dočasná proměnná (užitečné pro optimalizaci).

Jde o linearizovanou podobu syntaktického stromu.

Překlad výrazu x+y*z na tříadresový kód:

t1 := y * z

t2 := x + t1

Další formy tříadresových instrukcí: s unárním operátorem (x = -y), *copy* instrukce (x = y), indexované copy instrukce (x = y[0]), nepodmíněný skok (goto L), podmíněný skok (if x goto L), volání procedur (call p, n; předtím uvedeno *n* parameterů).

Trojice a čtveřice

Implementací tříadresového kódu jsou záznamy se třemi nebo čtyřmi poli: trojice resp. čtveřice. Následující příklady budou ukázány na výrazu: a := b * (-c) + d [b]

Čtveřice

Záznam má čtyři položky nazývané **op, arg1, arg2 a res**. Tříadresový příkaz ve tvaru x := y op z je reprezentován umístěním op do op, y do arg1, z do arg2 a x do res. Některé tříadresové příkazy nepotřebují všechny položky (např. x := y).

Výhoda čtveřic oproti trojicím – v optimalizaci kompilátoru, kdy jsou instrukce často přemísťovány; přesun čtveřic je ok (nová pozice se dá hned určit podle dočasných proměnných), u trojic je při posunu třeba změnit reference na výsledky, protože jsou určeny svou pozicí.

Trojice

Jestliže se chceme vyhnout generování dočasných proměnných, je možné použít formu trojic. Trojice obsahuje op, arg1 a arg2. Místo dočasných proměnných jsou indexy do pole trojic (jejich pozice).

Příklady

Ukažme si tříadresový kód, čtveřice a trojice na příkladě výrazu:

a := b * (- c) + d [b]

Tříadresový kód

```
t1 := - c
t2 := b * t1
t3 := d [ b ]
t4 := t2 + t3
a := t4
```

Čtveřice

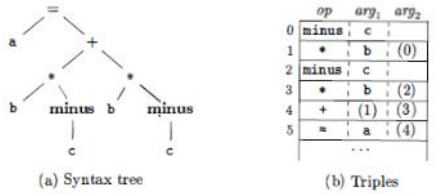
	op	arg1	arg2	res
(1)	uminus	c		t1
(2)	*	b	t1	t2
(3)	loadidx	d	b	t3
(4)	+	t2	t3	t4
(5)	:=	t4		a

Trojice

	op	arg1	arg2
(1)	uminus	c	
(2)	*	b	(1)
(3)	loadidx	d	b
(4)	+	(2)	(3)
(5)	:=	a	(4)

Překlad

u čtveřic voláme metodu pro generování instrukce a předáme jí jen parametry instrukce - např. GADD(…) = generate ADD a parametry jsou levá a pravá strana + docasna promenna;
u trojic tam musí bejt rozhodovací tabulka a kód se generuje v závislosti na tom, co je aktuálně na stacku přímo se to jmenuje Rozhodovací tabulka COMP



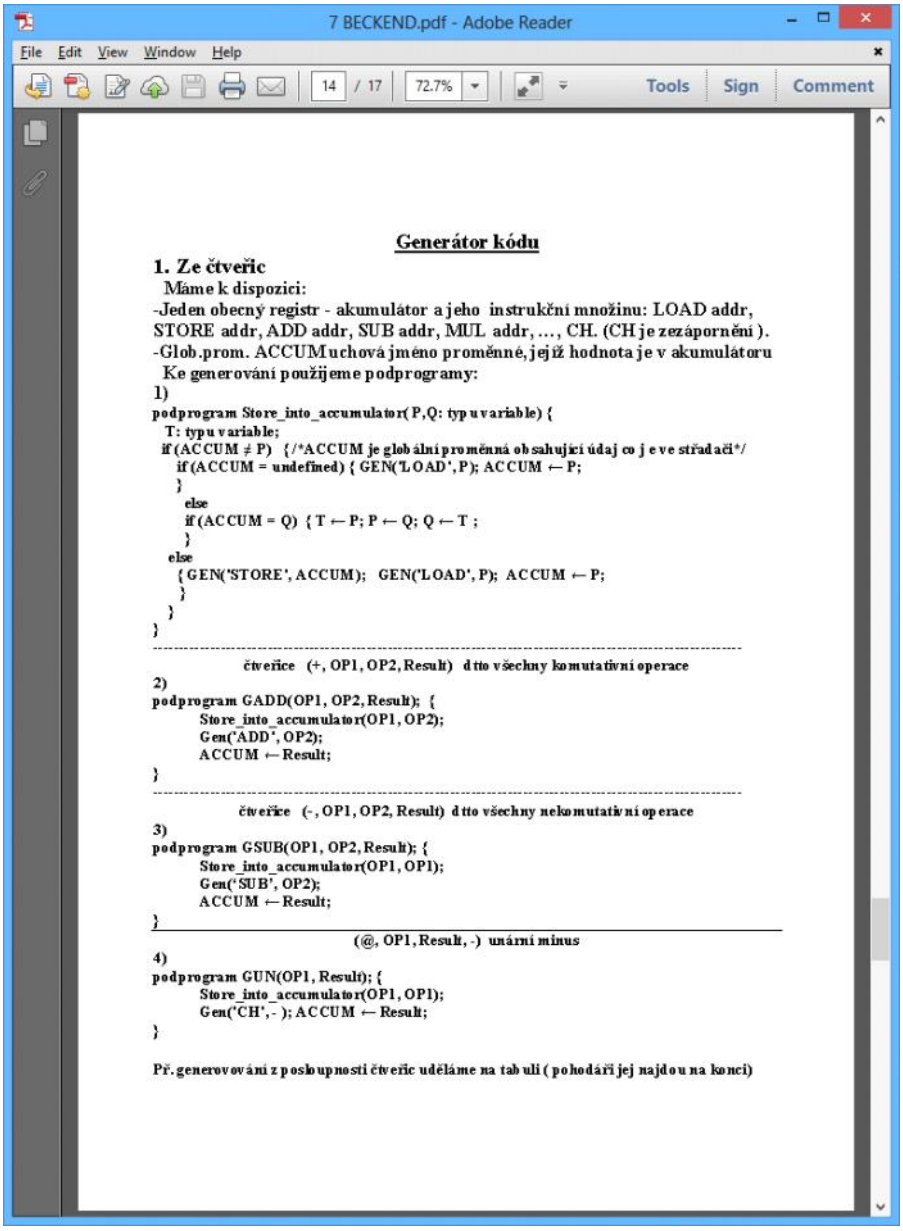
(a) Syntax tree

	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

(b) Triples

Figure 6.11: Representations of $a + a * (b - c) + (b - c) * d$

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>



Generátor kódu

1. Ze čtveřic

Máme k dispozici:

-Jeden obecný registr - akumulátor a jeho instrukční množinu: LOAD addr, STORE addr, ADD addr, SUB addr, MUL addr, ..., CH. (CH je zezápornění).

-Glob.prom. ACCUM uchová jméno proměnné, jejíž hodnota je v akumulátoru

Ke generování použijeme podprogramy:

1)

```
podprogram Store_into_accumulator( P,Q: typu variable) {
  T: typu variable;
 if (ACCUM ≠ P)  {/*ACCUM je globální proměnná obsahující údaj co je ve střadači*/
    if (ACCUM = undefined) { GEN('LOAD', P); ACCUM ← P;
    }
      else
      if (ACCUM = Q) { T ← P; P ← Q; Q ← T ;
      }
   else
    { GEN('STORE', ACCUM);  GEN('LOAD', P);  ACCUM ← P;
    }
  }
}
```

čtveřice (+, OP1, OP2, Result) dtto všechny komutativní operace

2)

```
podprogram GADD(OP1, OP2, Result); {
        Store_into_accumulator(OP1, OP2);
        Gen('ADD', OP2);
        ACCUM ← Result;
}
```

čtveřice (-, OP1, OP2, Result) dtto všechny nekomutativní operace

3)

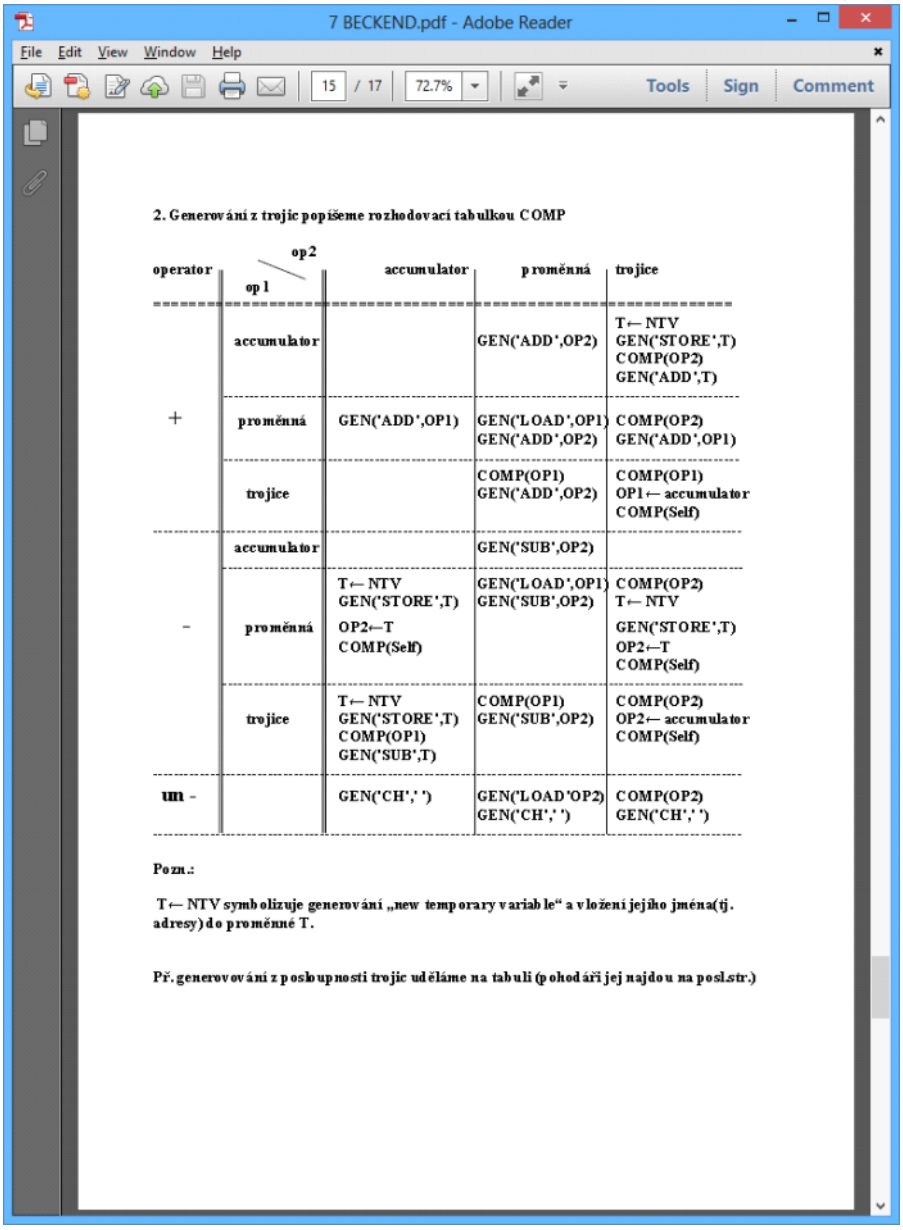
```
podprogram GSUB(OP1, OP2, Result); {
        Store_into_accumulator(OP1, OP1);
        Gen('SUB', OP2);
        ACCUM ← Result;
}
```

(@, OP1, Result, -) unární minus

4)

```
podprogram GUN(OP1, Result); {
        Store_into_accumulator(OP1, OP1);
        Gen('CH',- ); ACCUM ← Result;
}
```

Př. generovování z posloupnosti čtveřic uděláme na tabuli (pohodářiji najdou na konci)



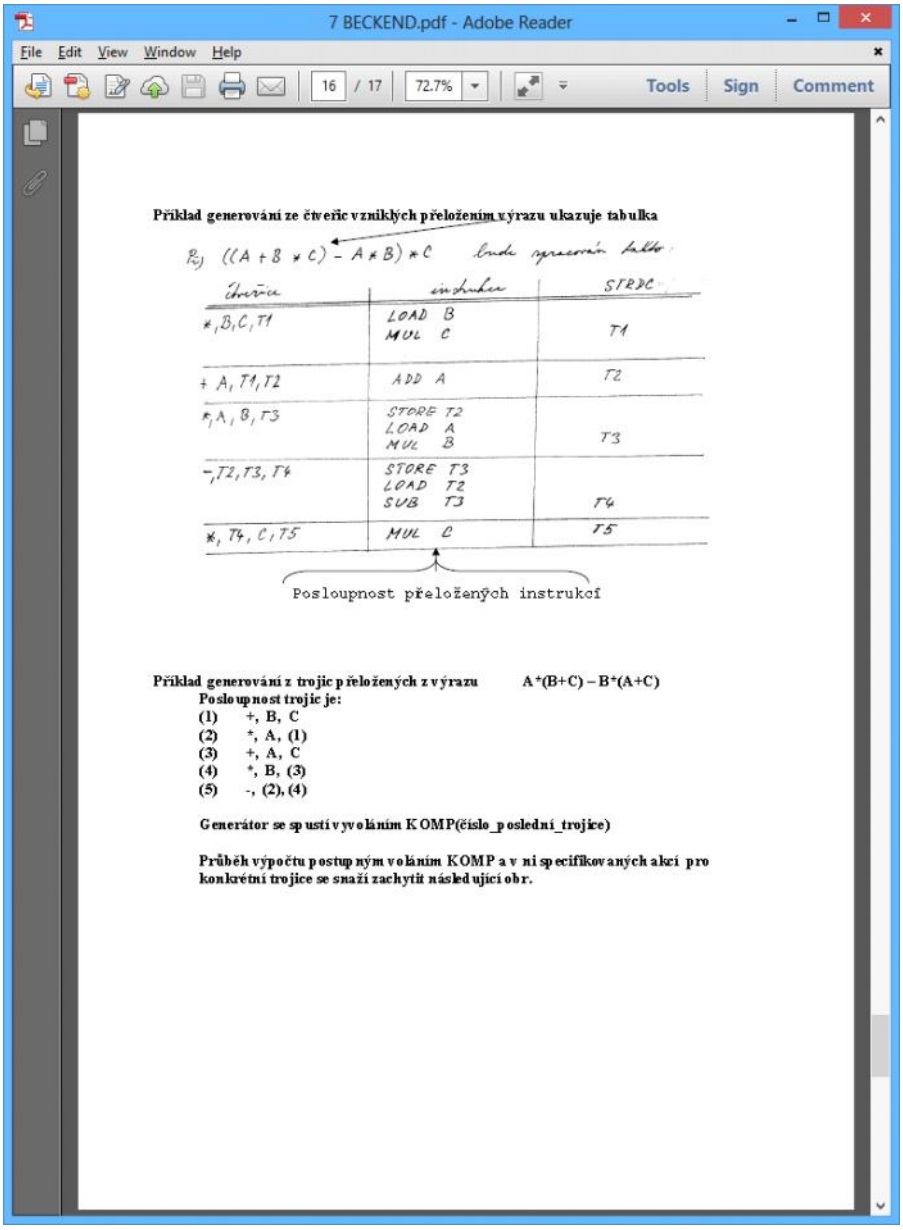
2. Generování z trojic popíšeme rozhodovací tabulkou COMP

operator	op1 \ op2	accumulator	proměnná	trojice
+	accumulator		GEN('ADD',OP2)	T← NTV GEN('STORE',T) COMP(OP2) GEN('ADD',T)
	proměnná	GEN('ADD',OP1)	GEN('LOAD',OP1) GEN('ADD',OP2)	COMP(OP2) GEN('ADD',OP1)
	trojice		COMP(OP1) GEN('ADD',OP2)	COMP(OP1) OP1← accumulator COMP(Self)
-	accumulator		GEN('SUB',OP2)	
	proměnná	T← NTV GEN('STORE',T) OP2←T COMP(Self)	GEN('LOAD',OP1) GEN('SUB',OP2)	COMP(OP2) T← NTV GEN('STORE',T) OP2←T COMP(Self)
	trojice	T← NTV GEN('STORE',T) COMP(OP1) GEN('SUB',T)	COMP(OP1) GEN('SUB',OP2)	COMP(OP2) OP2← accumulator COMP(Self)
un -		GEN('CH',' ')	GEN('LOAD'OP2) GEN('CH',' ')	COMP(OP2) GEN('CH',' ')

Pozn.:

T← NTV symbolizuje generování „new temporary variable" a vložení jejího jména(tj. adresy) do proměnné T.

Př. generovování z posloupnosti trojic uděláme na tabuli (pohodáří jej najdou na posl.str.)



Příklad generování ze čtveřic vzniklých přeložením výrazu ukazuje tabulka

Př.) ((A + B * C) - A * B) * C (vnitřní reprezentace kódu)

čtveřice	instrukce	STRDC
*, B, C, T1	LOAD B MUL C	T1
+, A, T1, T2	ADD A	T2
*, A, B, T3	STORE T2 LOAD A MUL B	T3
-, T2, T3, T4	STORE T3 LOAD T2 SUB T3	T4
*, T4, C, T5	MUL C	T5

Posloupnost přeložených instrukcí

Příklad generování z trojic přeložených z výrazu A*(B+C) – B*(A+C)

Posloupnost trojic je:

(1) +, B, C
(2) *, A, (1)
(3) +, A, C
(4) *, B, (3)
(5) -, (2), (4)

Generátor se spustí vyvoláním KOMP(číslo_poslední_trojice)

Průběh výpočtu postupným voláním KOMP a v ní specifikovaných akcí pro konkrétní trojice se snaží zachytit následující obr.

Tabulka symbolů – obsah, způsob manipulace při vytváření a využívání při překladu

Thursday, May 30, 2013 8:42 AM

Jakmile syntaktický analyzátor najde určitou konstrukci symbolů, tedy frázi, je třeba této konstrukci přiřadit význam. Součástí syntaktického analyzátoru bývá procedura (nebo více procedur či funkci), která je postupně pro každou frázi volaná a jejím úkolem je doplnit údaje do tabulky symbolů nebo do interního kódu.

OBSAH

Do tabulky symbolů (tabulky objektů) **ukládáme postupně všechny objekty** - pojmenované **identifikátory** (které nejsou klíčovými slovy), **proměnné** nebo **konstanty**, **uživatelské datové typy**, **funkce**, **procedury**, návěští apod., na které v kódu narazíme. Pojem objekt zde budeme chápat obecněji než je obvyklé v teorii programování, bude to **prostě jakýkoliv identifikátor**, **který není klíčovým slovem** a lexikální analýza ho proto odlišila od jiných identifikátorů.

Zapisujeme zde obvykle název, typ, adresu, případně počáteční hodnotu objektu, počet a typ parametrů funkce a další informace potřebné při dalším překladu, ale také při provádění programu.

Tabulka symbolů může vypadat takto:

Název	Typ	Délka	Deklarováno	Adresa	Použito
`delky`	integer array 10	40 B	A	.....	N
`I`	byte	1 B	A	.....	A
`pocet`	integer	4 B	A	.....	N
`x1`	real	6 B	A	.....	N
`z1`	*nedefinováno*	0	N	0	A

V tabulce vidíme objekty délky (pole o délce 10 prvků, prvky jsou celá čísla), I, pocet a x1, které již byly deklarovány a objekt I také použit. Objekt z1 ještě nebyl deklarován, ale už je v kódu použit. V jazyce, který umožňuje pracovat pouze s deklarovanými proměnnými, se jedná o sémantickou chybu.

U každého typu objektu potřebujeme uchovávat různé druhy informací. Například u proměnné je to název, adresa, datový typ, velikost potřebné paměti apod., u funkce název, adresa, návratový typ, počet a typ jednotlivých parametrů, příp. zda jsou volány hodnotou nebo odkazem (jestliže jsou volány odkazem, musí sémantický analyzátor navíc ošetřit, aby ve volání funkce byly jako skutečné parametry použity pouze názvy proměnných a nikoli například výrazy nebo konstantní hodnoty), u dalších typů objektů to budou opět jiné údaje. Řádky tabulky mohou být navzájem závislé (jeden uživatelský datový typ může využívat deklaraci již dříve uvedeného, popř. proměnná je typu deklarovaného dříve, . . .), nesmí se však jednat o kruhovou závislost.

Tato tabulka nám slouží k mnoha účelům. **Využívá ji zejména sémantický analyzátor** (kontroluje, zda proměnná použitá v kódu je deklarovaná a zda její datový typ odpovídá jejímu použití, jestli u funkce souhlasí počet a typ argumentů, atd.), **používá se také u generování cílového kódu** (překladač musí vědět, kolik místa v paměti má vyhradit pro jednotlivé symboly).

Při interpretaci obvykle není nutné uchovávat informaci o adrese, samotná tabulka symbolů může sloužit jako úschovna symbolů, se kterou pak neustále pracujeme.

ZPŮSOB MANIPULACE PŘI VYTVÁŘENÍ A VYUŽÍVÁNÍ PŘI PŘEKLADU

Tabulka symbolů může být **vytvářena již lexikálním analyzátorem**, ten však má **omezené možnosti** při zjišťování některých údajů, proto je v mnoha případech vhodnější přenechat tuto práci syntaktickému nebo sémantickému analyzátoru. Často používaný postup je vytváření tabulky lexikálním analyzátorem (kdykoliv narazí na identifikátor, který není klíčovým slovem, uloží ho do tabulky) s tím, že **další části překladače doplňují zbývající informace o vlastnostech uloženého**

identifikátoru.

Otázkou je, **jak** vlastně **řadit** jednotlivé objekty v tabulce. Důležitým kritériem je rychlost vyhledávání, protože k tabulce symbolů přistupuje zejména sémantický analyzátor velmi často. U jednodušších jazyků je možné tabulku automaticky řadit **podle abecedy**, u složitějších jazyků řešíme **indexací**, kdy zároveň s tabulkou vytváříme indexový seznam (příp. soubor), ve kterém jsou odkazy na objekty seřazené podle abecedy.

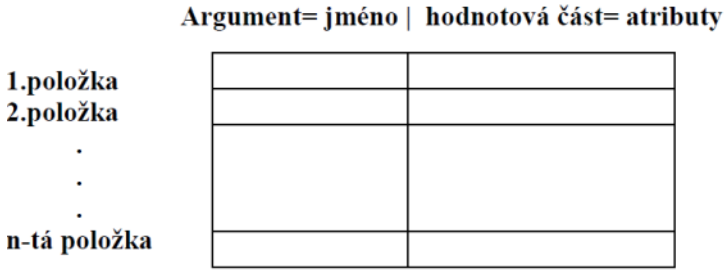
Speciální implementaci vyžaduje tabulka symbolů **pro jazyk s blokovou strukturou**, jako je třeba Pascal. Rozlišují se zde **lokální a globální objekty** a přístupnost lokálních je omezena. Každá proměnná je viditelná v tom bloku, ve kterém je deklarovaná, a také ve všech blocích vnořených. Když v určitém bloku použijeme proměnnou, hledáme informace o ní nejdřív v tom bloku, ve kterém se nacházíme. Při neúspěchu se posouváme do nadřízeného bloku a tak postupujeme, dokud ji nenajdeme. Pokud neuspějeme ani v hlavním bloku, znamená to, že byla použita proměnná, která není deklarovaná, jde o sémantickou chybu. **Každý blok má svoji vlastní tabulku**. S celou strukturou **se pracuje jako s** klasickým **zásobníkem**. Každá z tabulek má svou vlastní organizaci a je z ní přístupná nadřízená tabulka. „Aktivní tabulka je na vrcholu zásobníku, kde také začínáme prohledávat. Při vyhodnocení konce bloku se aktivní tabulka ze zásobníku odstraní. Po jejím odstraněni se sem přesune nadřízená tabulka. Tabulka hlavního bloku zůstává v zásobníku až do konce vyhodnocování programu, je odstraněna až jako poslední po vyhodnocení celého programu.

Tabulka symbolů

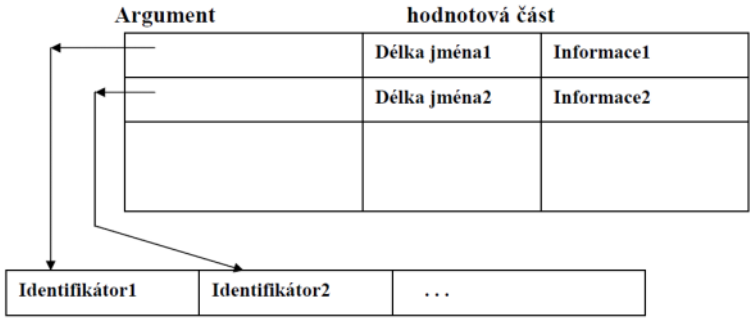
- uchovává informace o objektech
- umožňuje kontextové kontroly
- umožňuje operace
    a. inicializaci informace pro standardní jména
    b. vyhledání jména
    c. doplnění informace ke jménu
    d. přidání položky pro nové jméno
    e. vypuštění položky či skupiny položek

Struktura tabulky symbolů

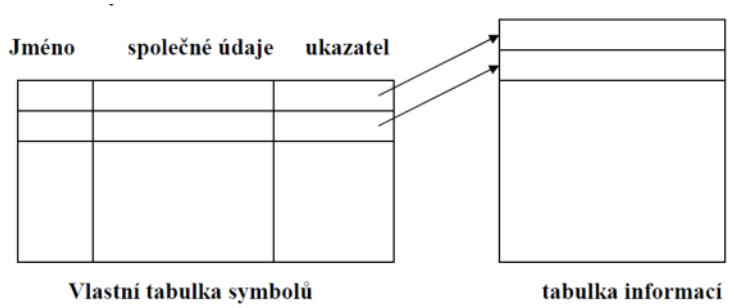
- s jednoduchou strukturou



- s oddělenou tabulkou identifikátorů

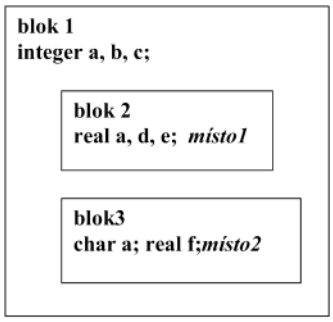


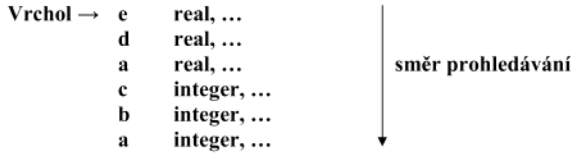
- s oddělenou tabulkou informací

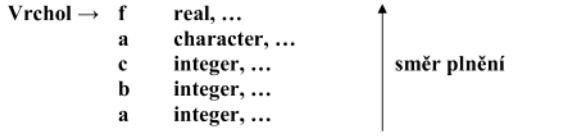


- uspořádané do podoby zásobníku

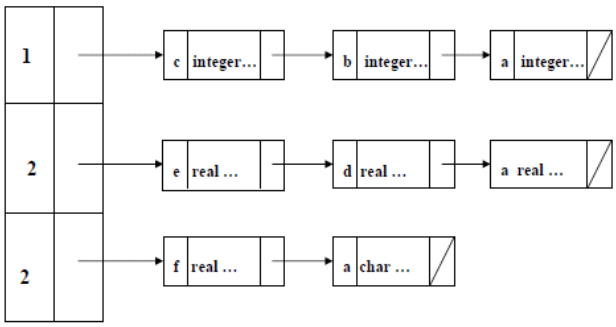
Tabulka symbolů uspořádaná do podoby zásobníku (pro jazyky s blokovou strukturou).
Rozsahová jednotka je blok, modul, funkce, balík,…
Respektuje zásady lokality



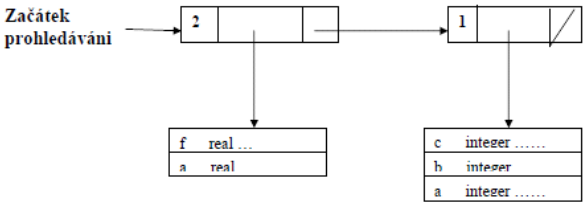




- s blokovou strukturou



Překládá-li se uvnitř bloku 3:



Implementace tabulky symbolů

- **Vyhledávací netříděné tabulky** (jen pro krátké programy)
  - prostá struktura
  - lineární seznam
- **Vyhledávací setříděné tabulky**
  - průběžné setřiďování
  - setřídění po zaplnění
- **Frekvenčně uspořádané tabulky**
- **Binární vyhledávací stromy**
- **Tabulky s rozptýlenými položkami**

Ukládání polí a struktur

Pole i struktury mají pevnou adresu začátku pole a pro přístup k jednotlivým prvkům se výsledná adresa dopočítává. Pole mohou být v paměti uložena buď po řádcích nebo po sloupcích. Tomu musí odpovídat mapovací funkce, která vypočítává relativní adresu prvků. K této adrese musí být připočtena adresa začátku pole.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Principy přidělování paměti překladačem

Thursday, May 30, 2013 8:43 AM

Přeložený program dostane od operačního počítače k dispozici blok paměti, který obecně může být rozdělen na následující části:

- Vygenerovaný cílový kód
- Statická data
- Řídící zásobník
- Hromada

Základní způsoby přidělování:

- **Statické** (přidělení paměti v čase překladu)
- **Dynamické** (přiděleno v run time) **- v zásobíku** nebo **na haldě**

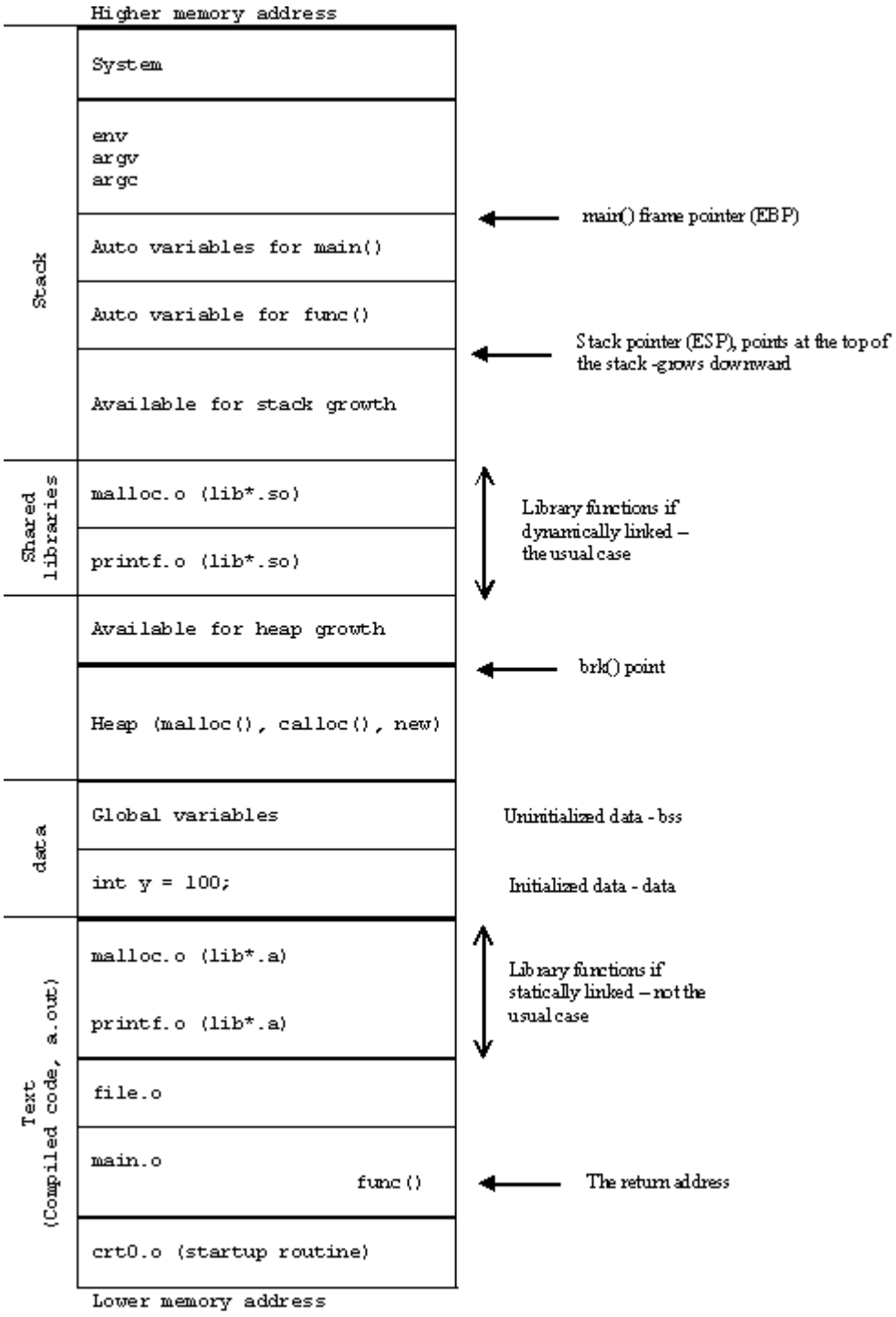
Velikost vygenerovaného kódu je známa již v době překladu, takže jej může překladač umístit **do staticky definované oblasti** (*Code/cílový kód programu*), obvykle na začátek přiděleného paměťového prostoru.

Rovněž velikost **statických datových objektů** může být známa již v době překladu a překladač je může **umístit za program** nebo uložit dokonce jako součást programu (to lze pouze u těch programovacích jazyků, které neumožňují rekurzivní volání procedur – Fortran).

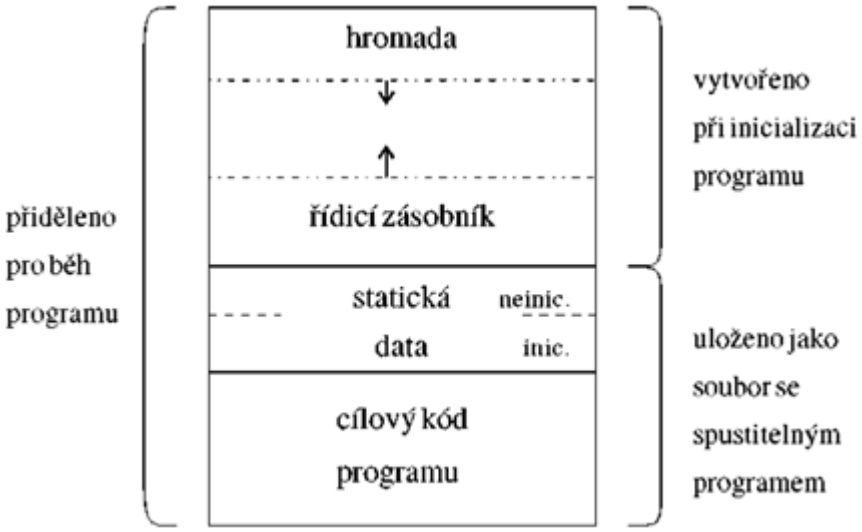
Jazyky umožňující rekurzi (Pascal, C, …) využívají pro aktivace podprogramů řídicího **zásobníku (stack)**, do kterého se ukládají jednotlivé ***aktivační záznamy*** (AZ jsou generovány při voláních procedur).

Pro účely **dynamického přidělování paměti** (explicitně vyžadovaného voláním příslušných funkcí nebo implicitně při přidělování paměti například pro pole s dynamickými rozměry) se používá zvláštní část paměti zvaná **hromada (heap)**.

Vzhledem k tomu, že se velikosti použité části paměti pro zásobník a hromadu v průběhu činnosti programu mohou značně měnit, je výhodné pro obě části využít opačné konce společné části paměti – viz obrázek. **Zásobník roste směrem k nižším adresám, hromada směrem k vyšším**. Nedostatek paměti se rozpozná tehdy, jestliže ukazatel konce některé oblasti překročí hodnotu ukazatele konce druhé oblasti.



From <http://www.tenouk.com/ModuleW_files/ccompilerlinker006.png>



Pro zmíněné datové oblasti se používají následující hlavní metody přidělování paměti:

- **Statické** přidělování paměti v době překladu
- **Dynamické** přidělování paměti za běhu programu:
  - Přidělování paměti na *zásobníku*
  - Přidělování paměti z *hromady*

Statické přidělování paměti v době překladu

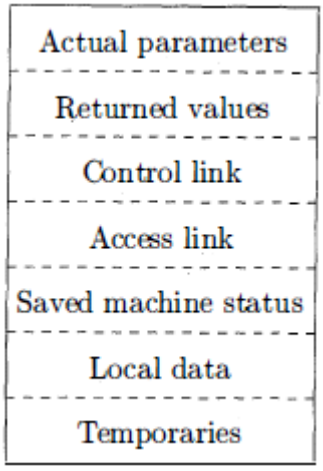
Při statickém přidělování paměti jsou všem objektům v programu **přiděleny adresy již v době překladu (globální data)**. Při kterémkoliv volání podprogramu jsou jeho lokální proměnné vždy na stejném místě, což umožňuje zachovávat hodnoty lokálních proměnných nezměněné mezi různými aktivacemi podprogramu. Statická alokace proměnných však klade na zdrojový jazyk určitá omezení. Údaje o velikosti a počtu všech datových objektů musejí být známy již v době překladu, **rekurzivní podprogramy mají velmi omezené možnosti**, neboť všechny aktivace podprogramu sdílejí tytéž proměnné, a konečně nelze vytvářet dynamické datové struktury.

Přidělování na zásobníku

Přidělování paměti pro aktivační záznamy na zásobníku se používá běžně u jazyků, které **umožňují rekurzivní volání podprogramů** nebo které **používají staticky do sebe zanořené podprogramy**. **Paměť pro lokální proměnné je přidělena při aktivaci podprogramu vždy na vrcholu zásobníku a při návratu je opět uvolněna**. To ale zároveň znamená, že hodnoty lokálních proměnných se **mezi dvěma aktivacemi podprogramu nezachovávají**.

Aktivace procedur při běhu programu jde znázornit ***aktivačním stromem***. Co uzel, to jedna aktivace, kořen je aktivací hlavní procedury, která je volána po spuštění programu; potomci uzlu *p* = volání procedur z procedury *p*. Kořen aktivačního stromu je na dně zásobníku, poslední aktivace má svůj záznam na vrcholu zásobníku.

Každá „živá“ aktivace má **aktivační záznam**. Obsah aktivačního záznamu se liší podle implementovaného jazyka.



1. ***dočasné hodnoty*** – vypadnou po vyhodnocení výrazů, jsou tu, pokud nemohou být udržovány v registrech
2. ***lokální data*** – patřící k dané proceduře s příslušným aktivačním záznamem
3. ***uložený strojový status*** – info o stavu stroje před voláním procedury. Typicky jde o:
   - *návratovou adresu* (= hodnota program counteru, kam se má pak procedura vrátit) a o
   - *obsah registrů* použitých procedurou (musí být obnoveny po návratu z procedury)
4. ***access link = statický ukazatel*** – pro lokaci dat, která procedura potřebuje, ale která se nachází v jiném aktivačním záznamu
5. ***control link*** – ukazuje na aktivační záznam volajícího (caller)
6. ***vrácené hodnoty*** – prostor pro návratovou hodnotu volané funkce (kvůli rychlosti lepší dávat do registru)
7. ***vlastní parametry*** – parametry použité volající procedurou; pokud je to možné, jsou umístěny radši v registrech kvůli výkonnosti.

Při implementaci přidělování paměti na zásobníku bývá **jeden registr vyhrazen jako ukazatel na začátek aktivačního záznamu na vrcholu zásobníku**. Relativně k tomuto registru se pak počítají všechny adresy datových objektů, které jsou umístěny v aktivačním záznamu. Naplnění registru a přidělení nového aktivačního záznamu je součástí volací posloupnosti, obnovení stavu před voláním se provádí během návratové posloupnosti.

Volací (a návratové) posloupnosti se od sebe v různých implementacích liší. Jejich činnost bývá rozdělena mezi volající a volaný program. Obvykle volající program určí adresu začátku nového aktivačního záznamu (k tomu potřebuje znát velikost záznamu vlastního), přesune do něj předávané argumenty a spustí volaný podprogram zároveň s uložením návratové adresy do určitého registru nebo na známé místo v paměti. Volaný podprogram nejprve uschová do svého aktivačního záznamu stavovou informaci (obsahy registrů, stavové slovo procesoru, návratovou adresu), inicializuje svá lokální data a pokračuje zpracováním svého těla. Při návratu opět volaný podprogram uloží hodnotu výsledku do registru nebo do paměti, obnoví uschovanou stavovou informaci a provede návrat do volajícího programu. Ten si převezme návratovou hodnotu a tím je volání podprogramu ukončeno.

přístup k nelokálním proměnným při statickém =lexikálním rozsahu platnosti jmen. To řeší tzv. **řetězec statických ukazatelů** (*access links*). Pro zrychlení přístupu k nelokálním proměnným se zavádí vektor ukazatelů – **displej**. Zamezí se tak průchod aktivačními záznamy pro hluboko zanořené podprogramy.

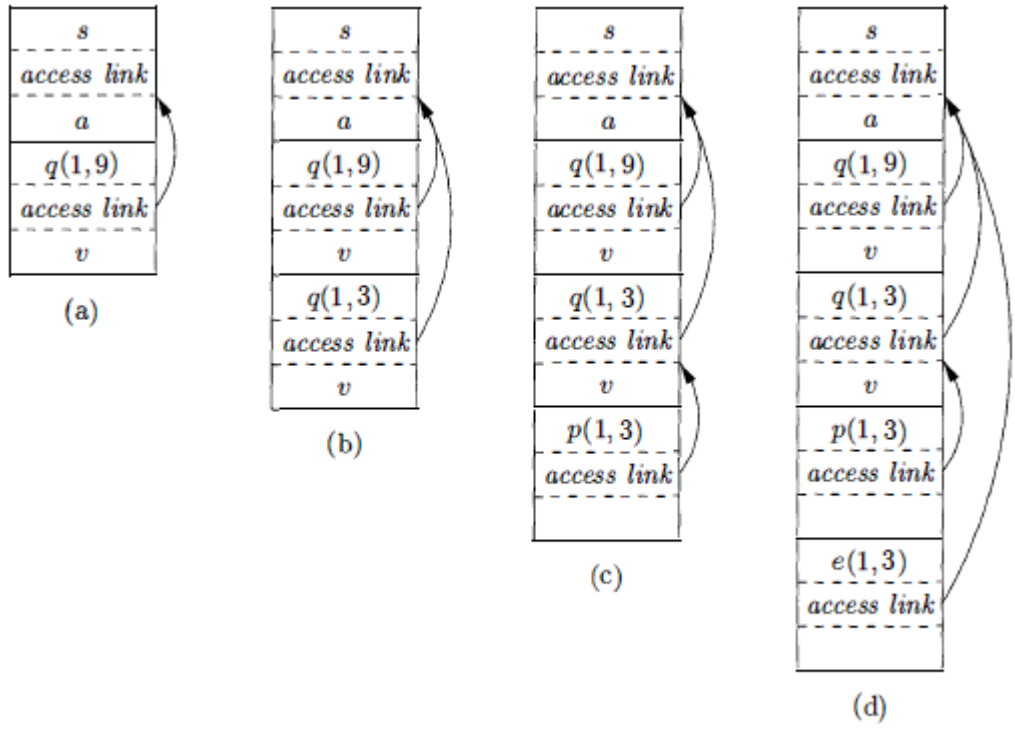


Figure 7.11: Access links for finding nonlocal data

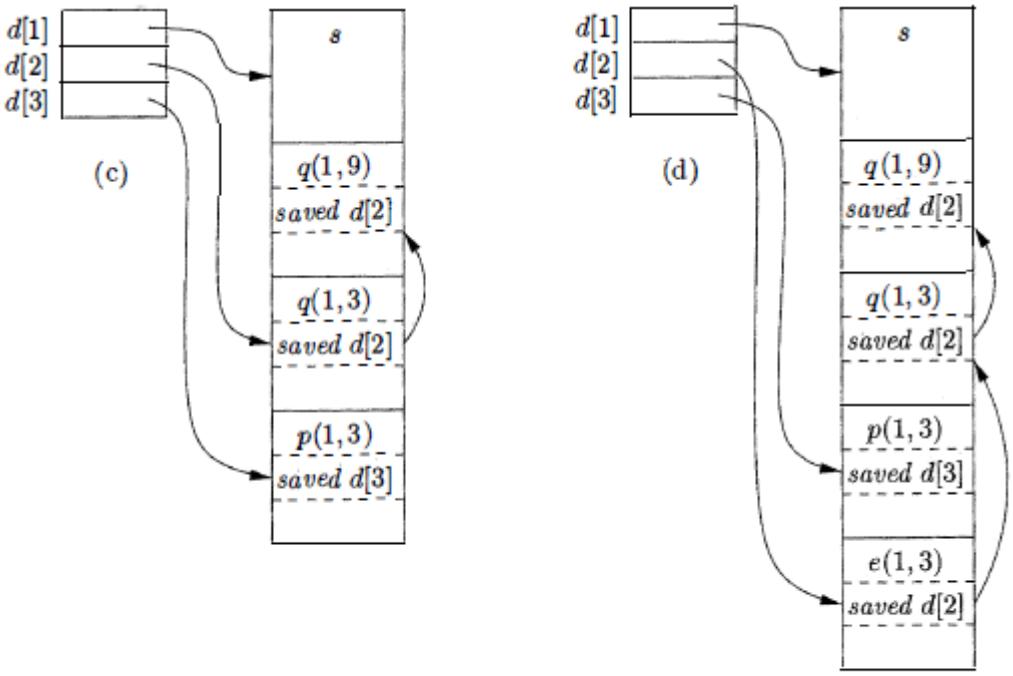


Figure 7.14: Maintaining the display

Kompilátor musí v čase kompilace určit rozvržení aktivačních záznamů a vygenerovat kód, který korektně přistupuje na místa v aktivačním záznamu. Proto *musí* být rozvržení AZ a generátor kódu navrhovány společně.

Přidělování z hromady

Strategie přidělování na zásobníku je nepoužitelná, pokud mohou hodnoty lokálních proměnných přetrvávat i po ukončení aktivace, případně pokud aktivace volaného podprogramu může přežít aktivaci volajícího. V těchto případech přidělování a uvolňování aktivačních záznamů se mohou překrývat, takže nemůžeme paměť organizovat jako zásobník. Aktivační záznamy se mohou v těchto nejobecnějších situacích přidělovat z volné oblasti paměti (hromady), která se jinak používá pro dynamické datové struktury vytvářené uživatelem. Přidělené aktivační záznamy se uvolňují až tehdy,

pokud se ukončí aktivace příslušného podprogramu nebo pokud už nejsou lokální data potřebná.

Při použití této strategie se pro vlastní přidělování a uvolňování paměti používají stejné techniky jako pro dynamické proměnné.

Správce paměti (memory manager) alokuje a dealokuje místo na heapu. V důsledku toho může dojít k fragmentaci heapu (vznik malých, nesouvislých míst = *děr*). Strategie *best fit* = alokuj nejmenší vhodnou a dostupnou díru.

Garbage collection hledá místo na heapu, které se už nepoužívá a může být proto realokované pro uchovávání dalších dat (Java, C#). Automaticky uvolňuje již nepoužívané objekty z paměti.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Vlastnosti jazykových konstrukcí pro statický a pro dynamický způsob přidělování paměti

Thursday, May 30, 2013 8:43 AM

Přímo podle Ježka:

Pro jazykovou konstrukci, která umožňuje jen statické přidělování, je možné řešit alokovanou paměť už během překladu. U těch dynamických není předem známo, kolik paměti je třeba vyhradit, je nutné to provádět až během vykonávání programu.

Jsou rekurze (nevíme předem, kolikrát se zanoříme)
Dynamické proměnné (new)
Dynamické typy (pole, jejichž velikost je daná výrazem)

Řeknu si, že chci udělat vlastní programovací jazyk a chci vědět, jak ho mám navrhnout - je třeba si uvědomit, co od něj budu potřebovat. Pokud potřebuju např. rekurzi, musím ho navrhnout tak, aby podporoval dynamické typy.

U statického přidělování paměti si mohu dovolit dělat statické proměnné, metody bez rekurze apod.
U dynamického pak vše ostatní jako rekurzi apod.

http://service.felk.cvut.cz/courses/X36PJP/Skripta_prednasky.pdf !!!

Důležitá hlediska jazykových konstrukcí:

- Dynamické typy
- Dynamické proměnné
- Rekurze
- Konstrukce pro paralelní výpočty

Podstatný je rovněž způsob:

- Omezování existence entit v programu
- Předávání parametrů funkce (hodnotou, odkazem)
- Určování přístupu k nelokálním entitám
    - na základě statického vnořování rozsahových jednotek,
    - na základě dynamického vnoření rozsahových jednotek.

Statické přidělování paměti:

- Globální proměnné
- Statické proměnné
- Proměnné jazyka bez rekurze (i s blokovou strukturou) (možno staticky na zásobníku)

Dynamické přidělování paměti (na hromadě):

- Dynamické typy a proměnné (překladač neví v době překladu kolik jich bude potřebovat)
- Proměnné předávané odkazem
- Pointery v C pro dynamicky alokované proměnné

Předávání parametrů podprogramům

- hodnotou (C, C++, Java, C#) formální parametr podprogramu je lokální proměnnou (tohoto podprogramu) do níž se předá hodnota (proměnná je zkopírována do zásobníku podprogramu)
- odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# parametry označené ref) předá informaci o umístění skutečného parametru (předána adresa na proměnnou) - u Javy se všechno předává prostřednictvím hodnoty (pass-by-value) tj. v případě instancí tříd (objektů) dochází k předávání **adresy** objektu ???
- výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu

Dynamické přidělování v zásobníku

Aktivační záznam obsahuje místo pro:

- Lokální proměnné
- Parametry
- Návratovou adresu
- Funkční hodnotu (je-li podpr. funkcí)
- Pomocné proměnné (pro mezivýsledky)
- Další informace potřebné k uspořádání aktivačních záznamů

Statická typová kontrola – referenční prostředí podprogramů je definováno staticky, tj. při překladu zdrojového programu. Pro každou deklaraci je staticky vymezen rozsah platnosti, tj. část zdrojového kódu, ve kterém lze deklarované jméno použít. V podprogramu pak kromě lokálních jmen lze použít ta nelokání jména, do jejichž rozsahu platnosti je definice podprogramu vnořena. Statické referenční prostředí je často založeno na blokové struktuře programu.

Statické referenční prostředí je při překladu reprezentováno tabulkou symbolů. Překlad deklarace znamená rozšíření tabulky symbolů o nový záznam, při dosažení místa konce platnosti deklarace se záznam odstraní nebo skryje. Při překladu těla podprogramu překladač na základě tabulky symbolů pro každé jméno rozhodne, zda je či není v daném místě použitelné a jaký datový objekt (nebo jiný programový prvek) označuje. Tím se dosáhne vyšší bezpečnosti programu (nepoužitelné jméno je odhaleno již při překladu) i vyšší efektivity cílového programu.

Dynamická typová kontrola – např. Lisp, referenční prostředí podprogramů je definováno dynamicky. Dynamicky definované referenční prostředí **se nevytváří ani nekontroluje při překladu, ale až při provádění programu**. Při spuštění programu se vytvoří referenční prostředí tvořené vazbami jmen definovaných jazykem. Při každém vstupu do podprogramu se referenční prostředí rozšíří o vazby lokálních jmen podprogramu, při návratu z podprogramu se jeho lokální prostředí odstraní. Při provádění příkazů se pro každé jméno hledá jeho vazba.

Dyn. Definované ref. Prostředí snižuje bezpečnost programu i jeho efektivitu (význam jména se hledá při provádění programu). Jeho výhodou je jednoduchá sémantika – nenajde-li se jméno v lokálním prostředí podprogramu A, hledá se v lokálním prostředí podprogramu B, ze kterého byl podprogram A vyvolán, případně v lokálním prostředí podprogramu C, z čehož byl vyvolán podprogram B apod.

From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Metody přidělování paměti

Základní způsoby: -Statické (přidělění paměti v čase překladu)
-Dynamické (přiděleno v run time) — v zásobníku / na haldě

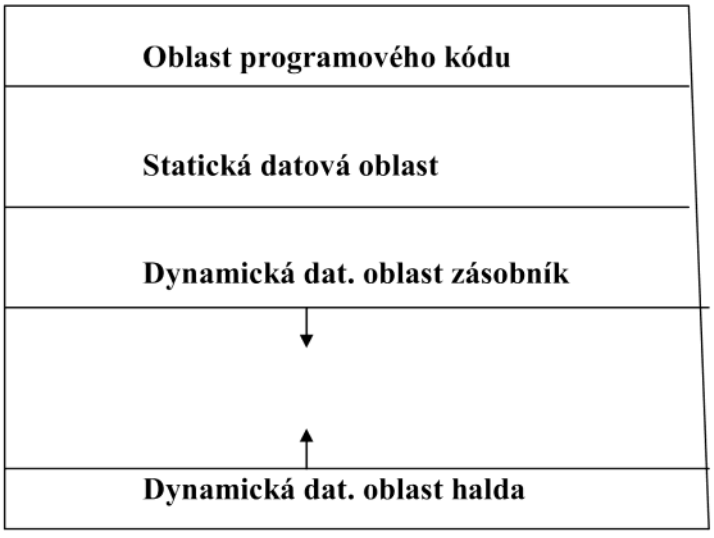
Důležitá hlediska jazykových konstrukcí:

- Dynamické typy
- Dynamické proměnné
- Rekurze
- Konstrukce pro paralelní výpočty

Podstatný je rovněž způsob:

- Omezování existence entit v programu (namespace, package, blok…)
- Určování přístupu k nelokálním entitám
  - na základě statického vnořování rozsahových jednotek,
  - na základě dynamického vnoření rozsahových jednotek.

Rozdělění paměti cílového programu



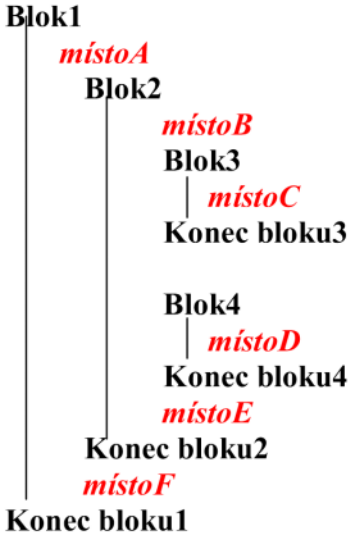
Page 2

Monday, June 3, 2013 1:08 PM

Statické přidělování paměti lze použít pro:

- **Globální proměnné**
- **Static proměnné**
- **Proměnné jazyka bez rekurze (i s vnořenou blokovou strukturou)**

Př.



Statické přidělování lze realizovat pomocí zásobníku

Ukažme obsah zásobníku v různých okamžicích výpočtu

Bloky

		3	4		
	2	2	2	2	
1	1	1	1	1	1
Místo A	**B**	**C**	**D**	**E**	**F**

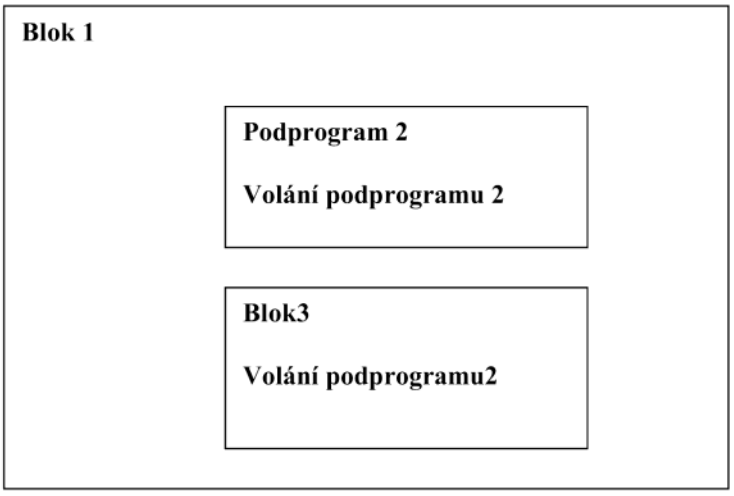
Vnořování podprogramů (funkcí, metod) je ale složitější, viz dále.

Dynamické přidělování v zásobníku

Část paměti přidělovaná při vstupu výpočtu do rozsahové jednotky programu se nazývá Aktivační Záznam (AZ představuje lokální prostředí výpočtu). Obsahuje místo pro:

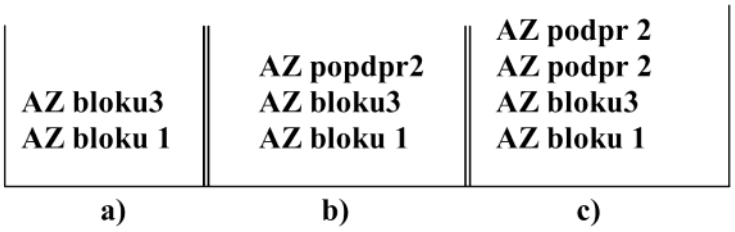
- **Lokální proměnné**
- **Parametry (je-li rozsahovou jednotkou podprogram či funkce)**
- **Návratovou adresu („ „)**
- **Funkční hodnotu (je-li rozsahová jednotka funkcí)**
- **Pomocné proměnné pro mezivýsledky (také možno v registrech)**
- **Další informace potřebné k uspořádání aktivačních záznamů**

Př.1

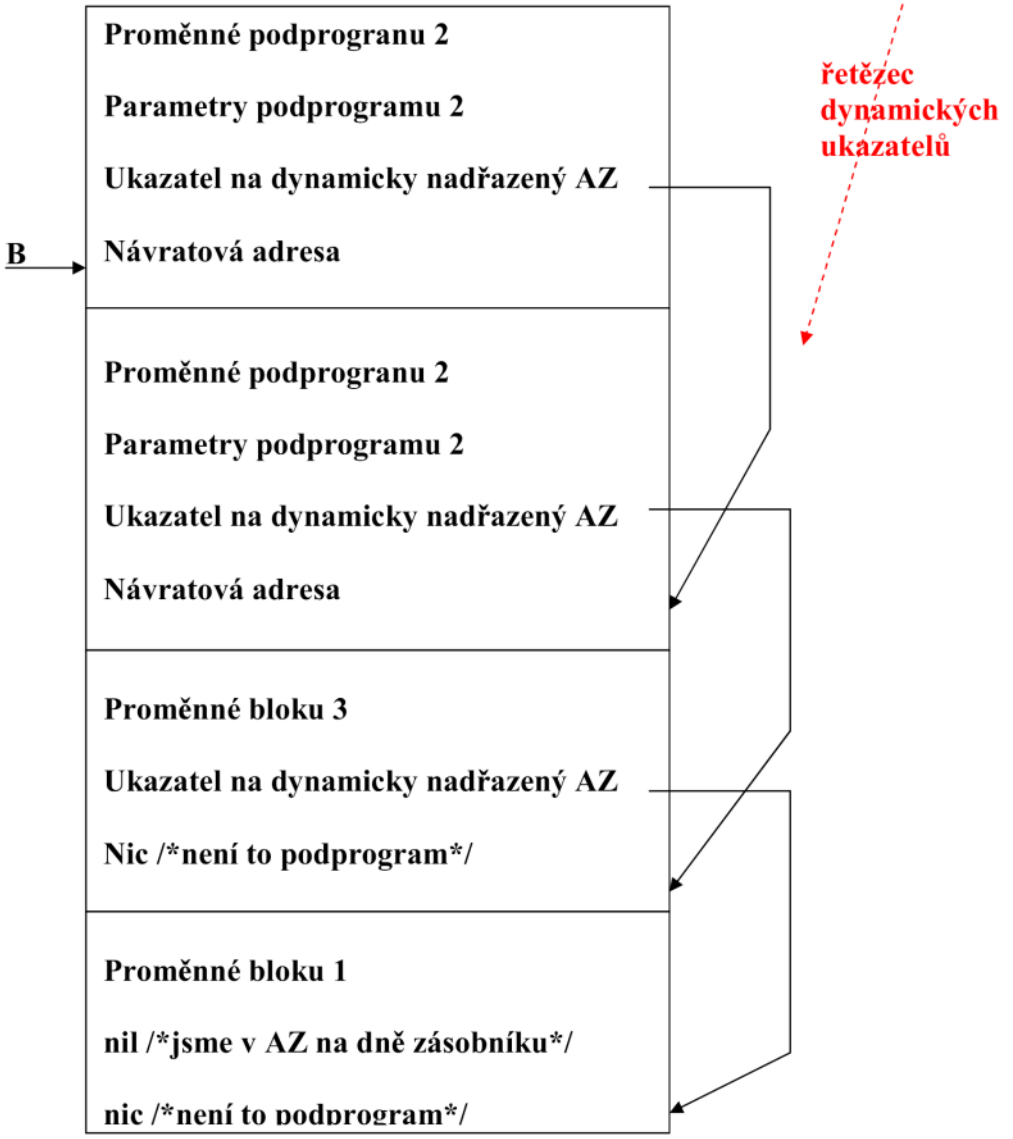


? stav výpočtového zásobníku v různých časech výpočtu

a) **Při vstupu do bloku 3**
b) **Při prvním volání podprogramu 2**
c) **Při rekurzivním vyvolání podprogramu 2**



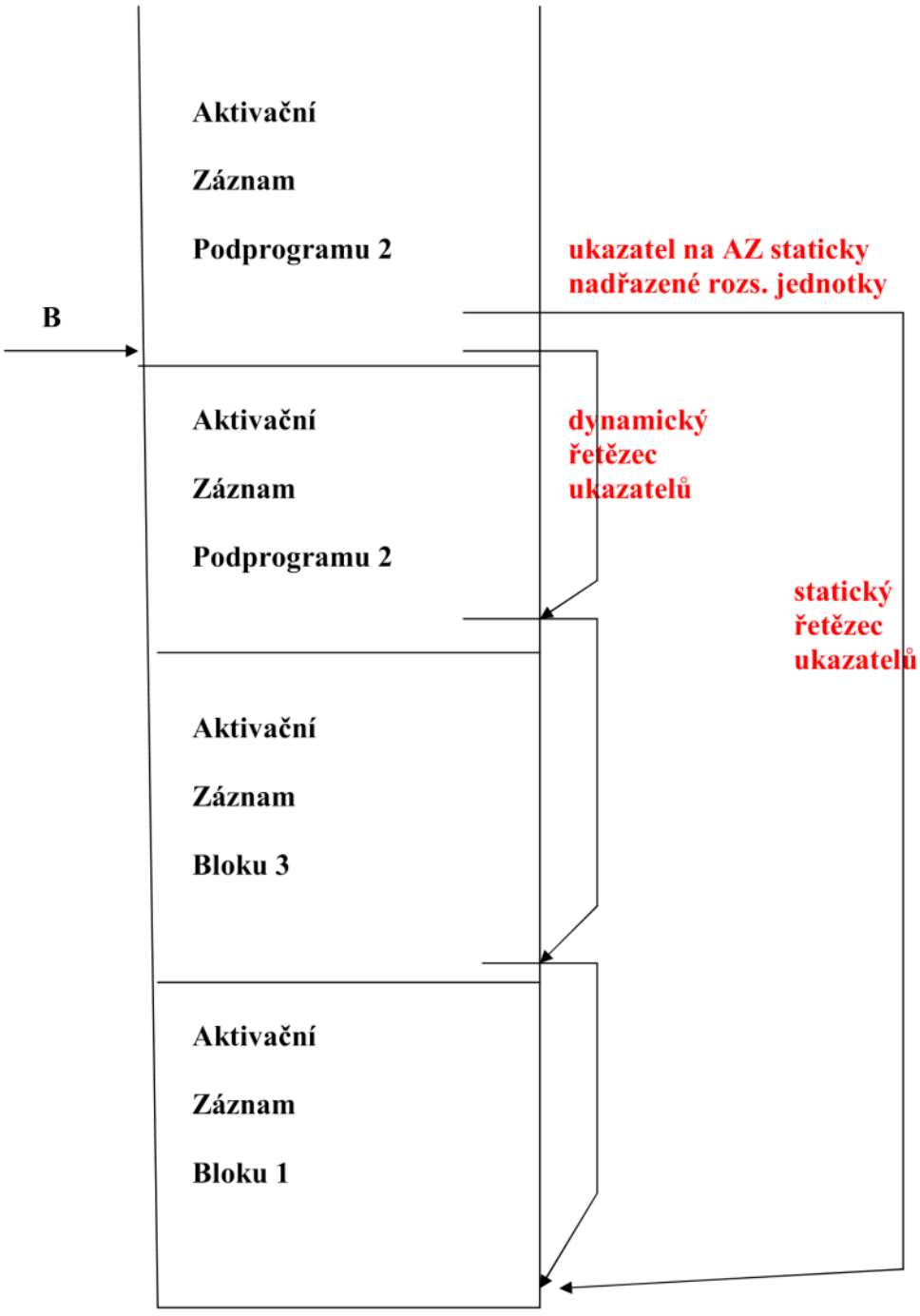
Jaké bude uspořádání aktivačních záznamů při rekurz. vyvolání podpr.2 ?
Pro rušení AZ při výstupu z jednotky potřebujeme tzv dynamický ukazatel



Obr. Zásobník při rekurzivním volání podprogramu 2

B je registr ukazující na vrcholový AZ

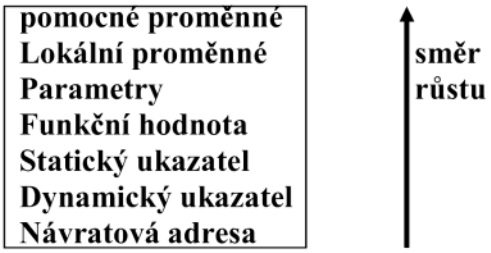
Potřebujeme ještě vyřešit přístup k nelokálním proměnným při statickém = lexikálním rozsahu platnosti jmen. To řeší tzv. řetězec statických ukazatelů



Obr. Zásobník se statickým (ukazuje na lexikálně nadřazený AZ) a dynamickým řetězcem ukazatelů při rekurzivním volání podprogramu 2
Pozn.: Statický uk. je nakreslen (pro přehlednost) jen u AZ rek. volání podpr.2

Vytváření řetězců ukazatelů

Nechť AZ má tvar:



Uvažujme zásobník Z, s vrcholem (nejvyšší zabranou adresou) T,
n je hladina deklarace , m je hladina volání
Při vstupu do rozsahové jednotky (vyvolání podprogramu nebo vstupu výpočtu do bloku = Aktivace rozsahové jednotky):

A1) Z[T + 1] ← návratová adresa /* pouze u podprogramů*/
A2) Z[T + 2] ← B /*nastavení dynamického ukazatele*/
A3) Z[T + 3] ← B
For i ← 1 to m – n do Z[T + 3] ← Z[Z[T + 3] + 2] /*nastavení statického ukazatele*/
A4) B ← T + 1 /*nastavení bázového registru*/
A5) T ← T + velikost aktivačního záznamu
A6) skok na první instrukci podprogramu a uložení do Z údajů o skutečných parametrech /*pouze u podprogramů*/

Pozn. Je-li podprogram překládán odděleně (neznámá velikost jeho AZ), pak je úprava T provedena až na začátku volaného podprogramu.

Při výstupu z rozsahové jednotky (Návrat z podprogramu nebo průchod koncem bloku):

N1) T ← B – 1
N2) B ← Z[B + 1]
N3) skok na adresu uloženou v Z[T + 1] /*pouze u podprogramů*/

Výstup z rozsahové jednotky nelokálním skokem (hladina n deklarace návěští je menší než hladina m místa s příkazem skoku)
Vždy platí n ≤ m

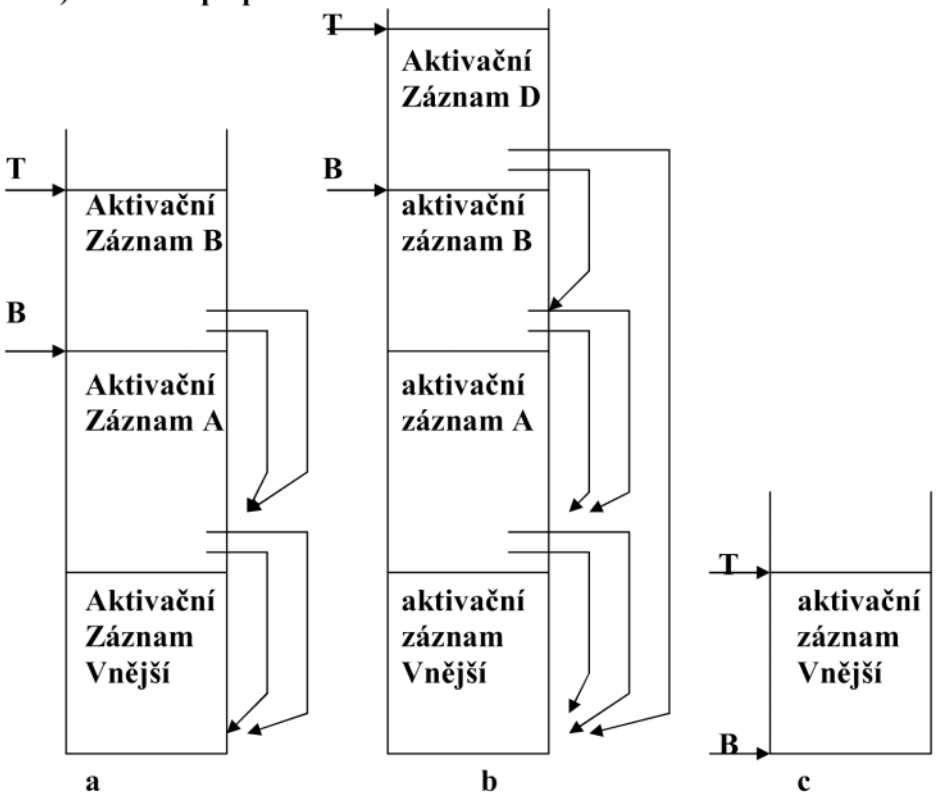
```
S1) for i ← 1 to m – n do { Pom ← B
                            repeat      T ← B – 1
                                        B ← Z[ B + 1]
                            until B ≠ Z[ POM + 2]
                          }
```

S2) skok na adresu, kterou návěští představuje

Př.2

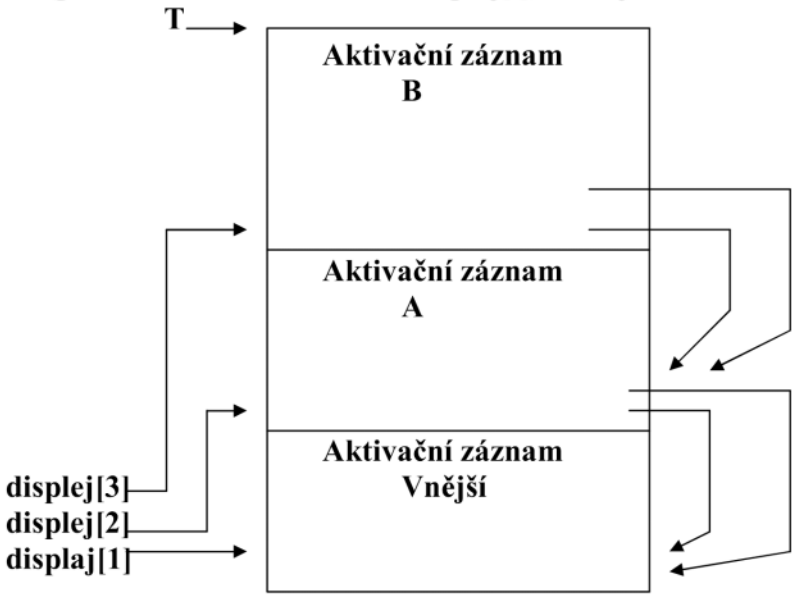
```
Podprogram Vnější //označme n hladinu deklarace
Label 1                    // a m hladinu místa použití
        Podprogram A
                Podprogram B
                Volání D  //zde  n=1; m=3
        Volání B
        Podprogram D
        Goto 1        //zde n=1; m=2
Volání A
1:
```

a) obsah Z při provádění B, před voláním D,
b) obsah Z po vyvolání D, ped provedením nelokálního skoku,
c) obsah Z po provedení skoku.

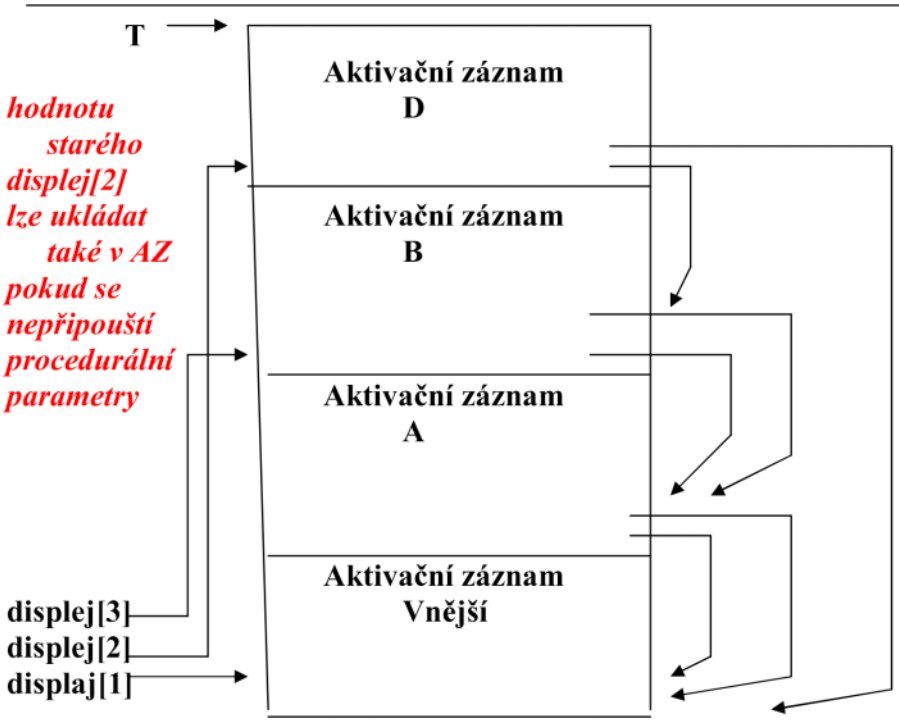


ad b) je to stav v okamžiku volání podpr. s hladinou deklarace 1, volaného v místě s hladinou 3
ad c) stav po výskoku z hladiny 2 do místa s hladinou 1

Zrychlení přístupu k nelokálním proměnným
(pomocí vektoru ukazatelů displej[i], kde i je hladina rozs. jedn.)



Obr. Stav Z při výpočtu B z př.2

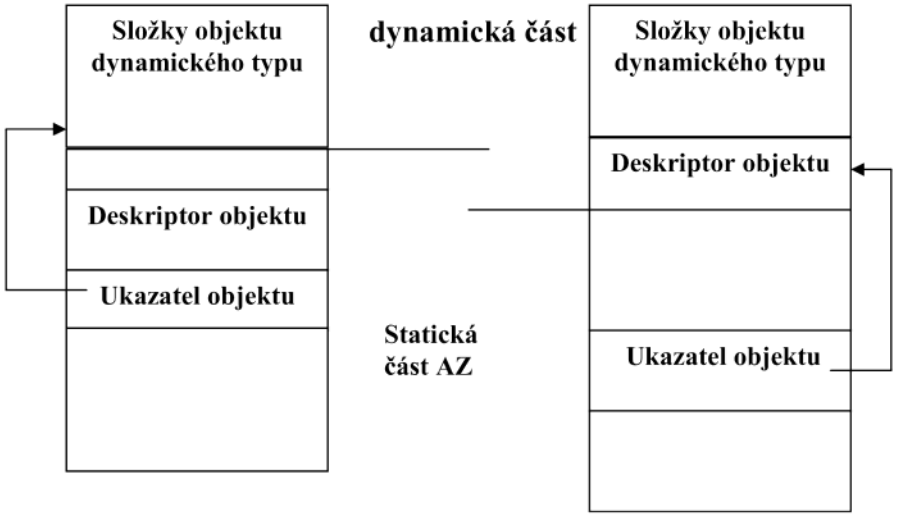


Obr. Stav Z při výpočtu D z př.2

Dynamická adresa proměnné je dvojice (n, p) = displej[n] + p

Objekty s dynamickými typy (typicky pole s proměnnými mezemi)

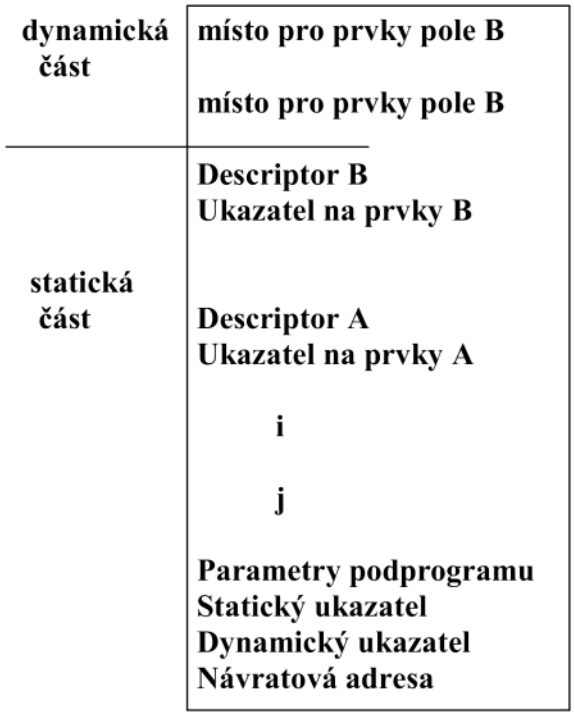
Možnosti struktury aktivačního záznamu s objektem dynamického typu



Deskriptor se vytvoří při překladu, uchovat se ale musí i při výpočtu

Př.3 Aktivačního záznamu s objekty dynamického typu

```
podprogram PRIKLAD;
        int i, j ;
        int A(m .. n);
        int B(p .. q, r .. s,  );
```



Předávání parametrů podprogramům

-hodnotou (C, C++, Java,C#) formální parametr je lokální proměnnou do níž se předá hodnota

-odkazem (C, C++ je-li parametrem pointer, objektové parametry Javy, C# označené ref) předá informaci o umístění skutečného parametru

-výsledkem - formální parametr je lokální proměnnou z níž se předá hodnota do skutečného parametru před návratem z podprogramu slouží jako výstupní parametr

-hodnotou výsledkem (novější Fortran) - kombinace

-jménem – má efekt textové substituce (jako historická zajímavost)

-v případě strukturovaných parametrů

-jsou-li to statické typy ⇒ předá se adresa prvého prvku

-jsou=li to dynamické typy ⇒ předá se ukazatel na descriptor

-je-li parametrem podprogram

-u jazyků nedovolujících hnízdění podprogramů ⇒předá se adresa začátku = pointer

- u jazyků dovolujících hnízdění podprogramů ⇒

spolu s adresou musí předdat i platné prostředí. Jsou různé možnosti co považovat za platné prostředí:

-**mělká vazba** ⇒ platné je prostředí v němž se nachází volání formálního podprogramu

-**hluboká vazba** ⇒platné je prostředí kde je předávaný podprogram definován

-**ad hoc vazba** ⇒platné je prostředí kde je vydán příkaz volání podprogramu jež má za parametr podprogram

Př.4

```
Podprogram  P1( ) {
        Prom x ;
        Podprogram P2 ( ) {
                Vytiskni (x) ; /*co se tady tiskne?*/
        };
        Podprogram P3 ( ) {
                Prom x ;
                x ← 3;
                P4(P2) ;
        };
        Podprgram P4( podprogram Px ) {
                Prom x ;
                x ← 4;
                call Px( );
        }
        x←1;
        P3( ) ;
}
```

prostředí, kde je předávaný podprogram definován

prostředí, kde je vydán příkaz volání s parametrem podprog.

prostředí, v němž je volán formální podprogram

Při mělké vazbě se tiskne … ?

Při hluboké vazbě se tiskne … ?

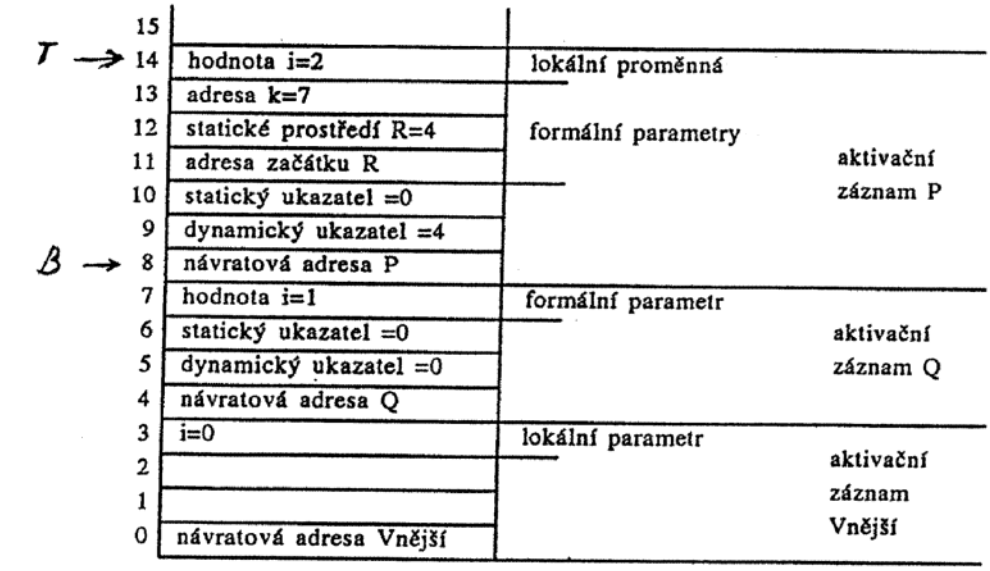
Při ad hoc vazbě se tiskne … ?

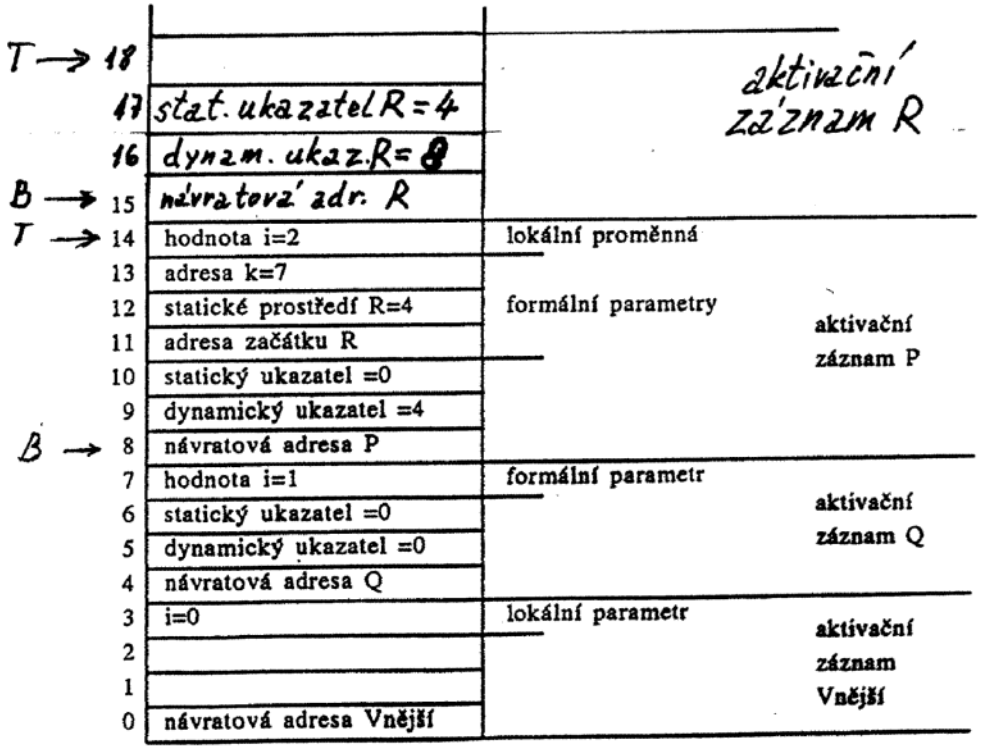
Př.5

Předpokládejme hlubokou vazbu. Co se vytiskne po spuštění procedury Vnější?

```
podprogram Vnejsi;  {
        prom i:int;
        podprogram P( podprogram FP; prom k:int;)  {
                prom i:int;
                i←k+1;  FP( ); tisk(i);
        }
        podprogram Q(i:int);
                podprogram R ( )  {
                        Tisk(i);
                }
                P(R,i);
        }
        i← 0; Q(i+1);
}
```

Stav před vyvoláním a po vyvolání formálního poprogramu FP z př.5





Přidělování paměti pro paralelní výpočty

Pro uložení AZ paralelního výpočtu nutno použít haldu nebo zobecněný zásobník

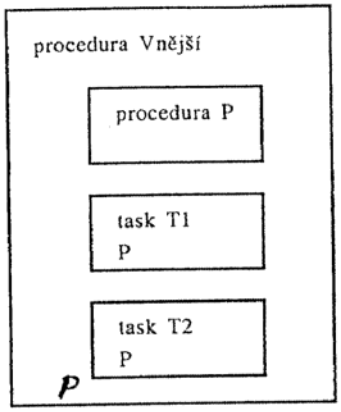
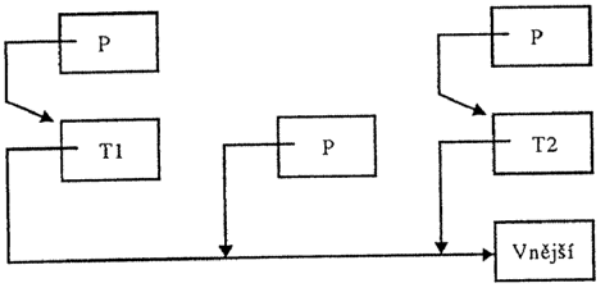
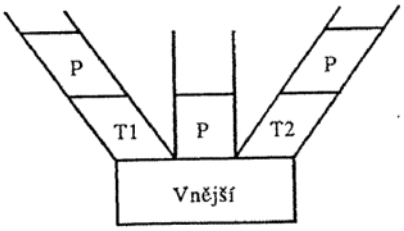


Schéma vnoření bloků programu

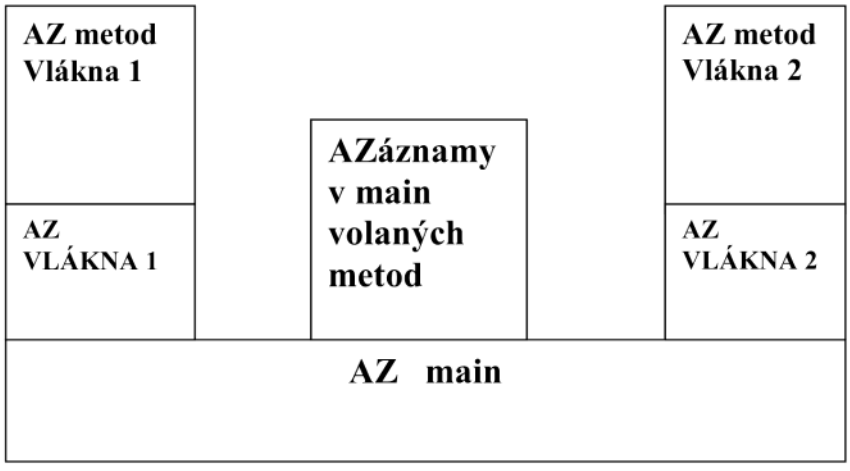


Struktura aktivačních záznamů při paralelním výpočtu



Uložení aktivačních záznamů ve zobecněném zásobníku

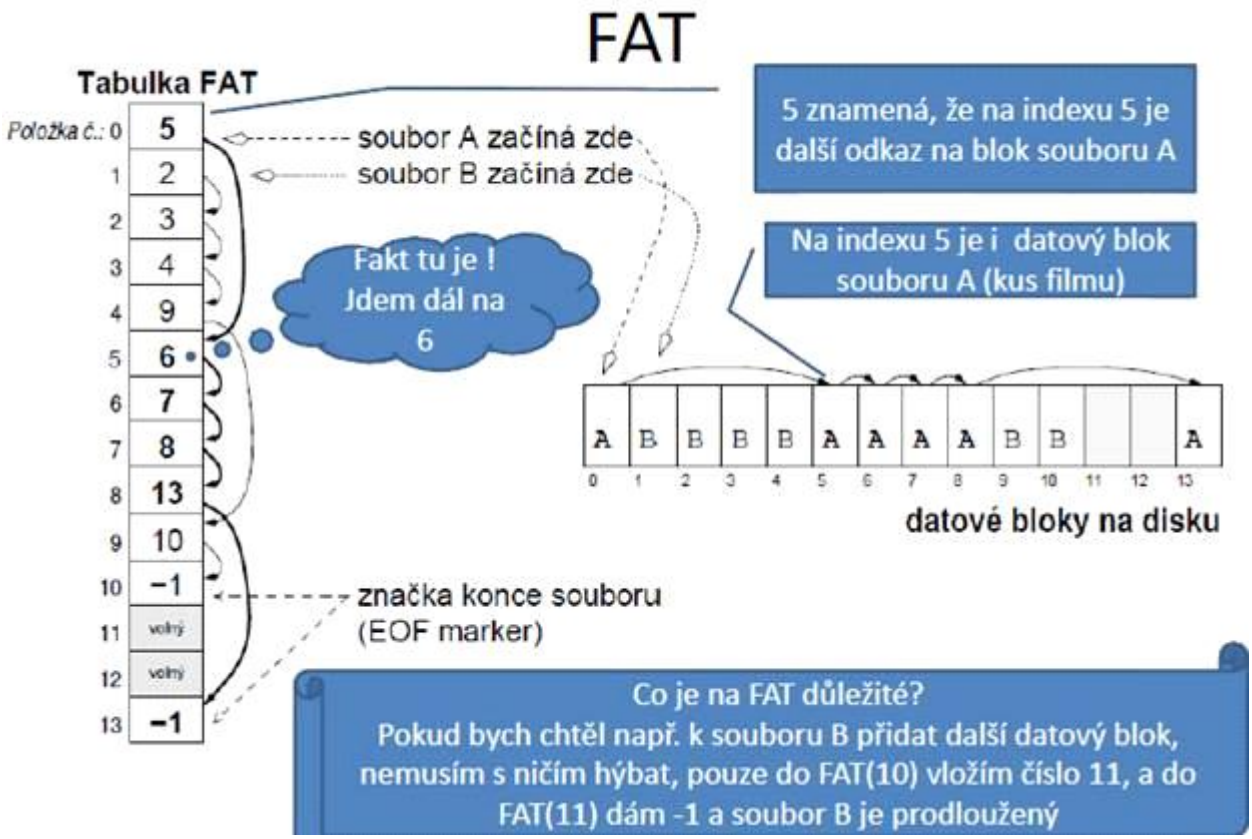
Př v javovském prostředí



filesystémy

Thursday, May 30, 2013 8:45 AM

FAT [ZOS SOUHRN]



From <https://d.docs.live.net/e3534876709763a3/Dokumenty/ZCU/Statnice/Statnice.docx>

Normalni formy

13. června 2013 10:08

Nultá normální forma (0NF) - tabulka v nulté normální formě obsahuje alespoň jeden sloupec (atribut), který může obsahovat více druhů hodnot.

První normální forma (1NF) - tabulka je v první normální formě, pokud všechny sloupce (atributy) nelze dále dělit na části nesoucí nějakou informaci neboli prvky musí být atomické. Jeden sloupec neobsahuje složené hodnoty.

Druhá normální forma (2NF) - tabulka je v druhé normální formě, pokud obsahuje pouze atributy (sloupce), které jsou závislé na celém klíči.

Třetí normální forma (3NF) - tabulka je ve třetí normální formě, pokud neexistují žádné závislosti mezi neklíčovými atributy (sloupci).

Čtvrtá normální forma (4NF) - tabulka je ve čtvrté normální formě, pokud sloupce (atributy) v ní obsažené popisují pouze jeden fakt nebo jednu souvislost.

Vektor přerušení

je adresa paměti handleru přerušení nebo **index pole tabulky vektorů přerušení, která obsahuje paměťové adresy přerušení**. Tabulka bývá většinou umístěna na začátku paměti, jak je běžné u procesorů architektury x86.

UML

Pro tvorbu diagramů systému, jejichž syntézou bude model, definuje např. **UML devět typů diagramů.**

1. Class Diagram
2. Component Diagram
3. Deployment Diagram
4. Object Diagram (Instance diagram)
5. Package Diagram
6. Composite Structure Diagram (popis vnitřní struktury třídy)
7. Use Case Diagram
8. Activity Diagram
9. State Machine Diagram
10. Sequence Diagram
11. Communication Diagram
12. Interaction Overview Diagram

Monitory

Sunday, June 2, 2013 9:17 PM

- Snaha najít primitiva vyšší úrovně, která zabrání části potenciálních chyb – například u semaforů lze operace P a V snadno zaměnit – vznik nežádoucích chyb…

Monitor je na rozdíl od semaforů jazyková konstrukce :

- Monitor je speciální typ modulu, ve kterém jsou sdružena data a procedury, které s nimi mohou manipulovat
- Procesy mohou volat procedury monitoru, ale nemohou přímo přistupovat k datům monitoru
- V monitoru může být v jednu chvíli aktivní pouze jeden proces; ostatní procesy jsou při pokusu o vstup do monitoru pozastaveny

Synchronizace procesů v monitoru:

- Monitory poskytují speciální typ proměnné nazývané podmínka (condition variable)
- Podmínky mohou být definovány a použity pouze uvnitř monitoru
- Podmínky nejsou proměnné v klasickém smyslu, tj. neobsahují hodnotu
- Podmínku lze chápat spíše jako odkaz na určitou událost nebo stav výpočtu

Nad podmínkami definovány operace wait a signal

- **c.wait**
  - volající bude pozastaven nad podmínkou c
  - pokud je některý proces připraven vstoupit do monitoru, bude mu to dovoleno
- **c.signal**
  - pokud existuje jeden, nebo více procesů pozastavených nad podmínkou c, reaktivuje jeden z pozastavených procesů, tj. bude mu dovoleno pokračovat v běhu uvnitř monitoru
  - pokud nad podmínkou nespí žádný proces, nedělá nic (na rozdíl od operace V(sem) nad semaforem, která si zapamatuje, že byla zavolána)

Použití: pro vzájemné vyloučení – KS programu zapouzdříme do procedury nebo fce a přesuneme do monitoru

Shrnutí + implementace: V monitoru může být aktivní pouze jeden proces, tedy na vstupu monitor pustí pouze jeden proces, ostatní čekají, až tento proces opustí monitor, poté může do něj vstoupit další (pozor – funkcionalitu lze rozšířit operacemi wait a signal).

Řešení problémů reakce na signal:

1. Hoare

Proces volající c.signal se POZASTAVÍ - vzbudí se až poté, co předchozí reaktivovaný proces opustí monitor nebo se pozastaví

2. Hansen

Signal smí být uveden pouze jako poslední příkaz v monitoru - po volání signal musí proces opustit monitor

3. Monitor v Javě

- S každým objektem je sdružen monitor, může být i prázdný
- Metoda nebo blok patřící do monitoru označena klíčovým slovem synchronized

```
class jmeno {
        synchronized void metoda() {
```

```
        ....
        }
}
```

S monitorem je sdružena jedna podmínka, metody:

- wait() – pozastavívolající vlákno
- notify() – označí jedno spící vlákno pro vzbuzení, vzbudí se, až volající opustí monitor (x c.signal, které pozastaví volajícího)
- notifyAll() – jako notify(), ale označí pro vzbuzení všechna spící vlákna

1. Implementace

```
mutex_lock:TSL R, mutex;; R:=mutex a mutex:=1
CMP R, 0;; byla v mutex hodnota 0?
JE ok ;; pokud byla na OK
CALL yield;; vzdáme se procesoru -naplánuje se jiné vlákno

JMP mutex_lock;; zkusíme znovu, pozdějí
ok: RET

mutex_unlock:
LD mutex, 0 ;; ulož 0 do mutex
RET
```

Voláním yield se dobrovolně vzdává daný proces procesoru ve prospěch jiných procesů. Jádro OS přesune proces mezi **připravené** a časem ho opět naplánuje

Problém uvíznutí (deadlock, zablokování) procesů, graf alokace zdrojů

Sunday, June 2, 2013 9:17 PM

- Např.:
  - Představme si „večeříci filozofy“ – vezmou pravou vidličku, ale nemohou vzít levou (protože tato je již obsazena sousedem).
  - Alokace HW: mějme vypalovačku CD (=R jako recorder) a scanner (S)
    - Dva procesy A a B chtějí nascanovat dokument a zapsat na CD ROM
    - A žádá R a dostane, B žádá S a dostane
    - A žádá S a čeká, B žádá R a čeká – uvíznutí
  - Totéž i se semafory: A provede P(R) a B provede P(S)…

Definice:

V množině procesů nastalo uvíznutí, jestliže každý proces množiny čeká na událost, kterou může způsobit pouze jiný proces množiny

Jak se projeví deadlock? :

- Například tak, že několik procesů je ve stavu blokovaný a nedostanou se z něj, ale ostatní procesy mohou být plánovány
- Např. semafor *s5* má hodnotu 0, a procesy P1, P2 zavolají oba operaci *P(s5)* – oba zůstanou viset ve stavu blokovaný, pokud žádný jiný proces se semaforem s5 nebude manipulovat

1. Podmínky vzniku uvíznutí

- Vzájemné vyloučení – každý zdroj přiřazen procesu
- Hold and wait – proces držící přiřazené zdroje může požadovat další zdroje
- Nemožnost odejmutí – (např. problém večeřících filozofů). Jednou přiřazené zdroje nemohou být procesu násilně odejmuty
- Cyklické čekání – musí být cyklický řetězec 2 nebo více procesů, kde každý z nich čeká na zdroj držený dalším členem

Pokud jedna z podmínek není splněna, uvíznutí nenastane.

2. Vyhladovění

Procesy požadují zdroje, musí existovat pravidlo pro jejich přiřazení. Může se stát, že některý process zdroj nikdy neobdrží, i když nenastalo uvíznutí!

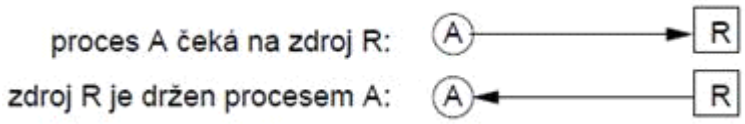
Např. “večeřící filosofové” v již uvedeném příkladu

- Každý zvedne levou vidličku, pokud je pravá obsazená tak levou položí
- Vyhladovění – pokud všichni zvedají a pokládají současně

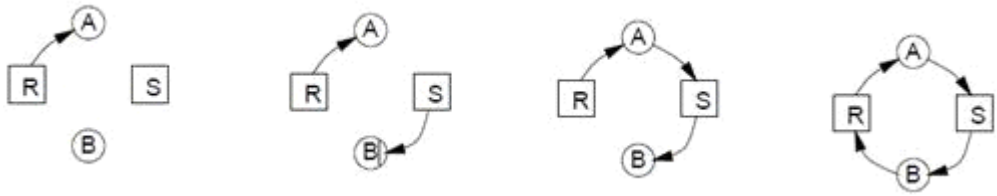
3. Graf alokace zdrojů

Podmínky vzniku uvíznutí mohou být modelovány pomocí orientovaných grafů = graf alokace zdrojů

- Graf má 2 typy uzlů
  - **Proces** – zobrazujeme jako kruh O
  - **Zdroj** – zobrazujeme jako čtverec []
- Význam hran
  - Hrana od zdroje k procesu: zdroj držen procesem
  - Hrana od procesu ke zdroji: proces je blokován čekáním na zdroj



Graf alokace zdrojů na přikladu s CD vypalovačkou (viz výše) – cyklus grafu je za uvedených podmínek výše nutnou a postačující podmínkou pro vznik uvíznutí:



4. Strategie zacházení s uvíznutím:

- Ignorování = „pštrosí algoritmus“
- Detekce a zotavení (odejmutí zdroje, zrušení změn, zrušení procesu)
- Dynamické zabrání (pečlivá alokace zdrojů)
- Prevence pomocí strukturální negace jedné z podmínek pro vznik uvíznutí

Klasické problémy meziprocesové komunikace – producent – konzument aj.

Sunday, June 2, 2013 9:18 PM

- Procesy komunikují:

1. Přes sdílenou paměť

- procesy na stejném uzlu
- někdy nevhodné kvůli bezpečnosti – globální data přístupná kterémukoliv procesu bez ohledu na semafor

2. Zasíláním zpráv

- Signál = speciální zpráva zaslaná procesu, obsahuje číslo signálu; je to asynchronní – proces jej ihned obslouží
- na stejném uzlu i na různých uzlech (komunikace po síti)
- zavedena 2 primitiva:
  - send(adresát,zpráva)
  - receive(odesilatel,zpráva)
- vlastnosti:
  - synchronizace:
    - blokující (synchronní)
    - neblokující (asynchronní)
    - většinou: send neblokující, receive blokující
  - adresování:
    - multicast = zpráva poslána skupině procesů
    - broadcast = zpráva poslána všem procesům
  - **problém** určení adresáta:
    - procesy mají pokaždé jiný PID => jak zjistit komu poslat zprávu -> řešení: adresujeme frontu zpráv
  - **další problémy:**
    - ztráta zprávy => potvrzení o přijetí, pokud vysílač nedostane potvrzení do nějaký doby (timeout), zprávu pošle znovu
    - ztráta potvrzení – práva došla ale ztratilo se potvrzení => číslování zpráv, duplicitní se ignorují
    - problém autentizace – ověřit, že nekomunikuje s podvodníkem: zprávy možno šifrovat – klíč známý autorizovaným uživatelům
- Mailbox = fronta zpráv využívaná více odesílateli a příjemci
- Port = omezená forma mailboxu – zprávy může vybírat pouze jeden příjemce

1. Lokální komunikace

- na stejném stroji => snížení režie na zprávy
  - Dvojí kopírování = z procesu odesílatele do fronty v jádře a z jádra do procesu příjemce
  - Rendezvous:
    - Eliminuje frontu zpráv – vyvolán send i receive – zprávu zkopírovat z odesílatele přímo do příjemce
  - Využitím virtuální paměti – paměť se zprávou je přemapována z odesílatele do procesu příjemce

2. Volání vzdálených procedur (RPC) = Remote Procedure Call

- Dovolit procesům (klentům) volat procedury umístěné v jiném procesu (serveru)
- Spojka(stub) klienta, serveru – spojky zakrývají, že volání není lokální

Klient zavolá spojku->spojková procedura zabalí argumenty do zprávy a pošle, a obdobně.

- Problémy RPC:
  - Parametry předané odkazem – posílání ukazatele nemá smysl
  - Není možné použití globálních proměnných
  - Reprezentace informace – stroje různých architektur

1. Problém soupeření a synchronizace
2. Kritické sekce a vzájemné vyloučení
3. Řešení vzájemného vyloučení
4. Semafory a jejich použití
5. Klasické synchronizační úlohy a jejich řešení
6. Příklady řešení v různých systémech

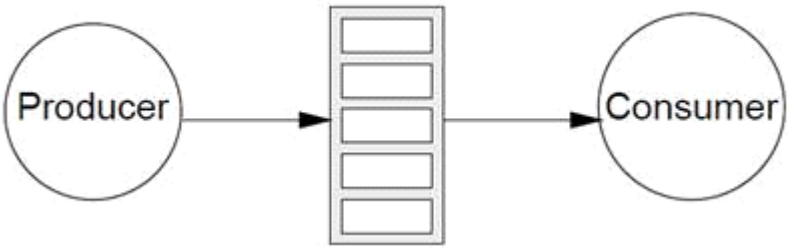
Podstata problému

- Souběžný přístup ke sdíleným datům může způsobit jejich nekonzistenci ⇒ nutná kooperace procesů
- Synchronizace běhu procesů
  - Čekání na událost vyvolanou jiným procesem
- Komunikace mezi procesy (IPC = *Inter-process Communication*)
  - Výměna informací (zpráv)
  - Způsob synchronizace, koordinace různých aktivit
  - Může dojít k uváznutí
    - Každý proces v jisté skupině procesů čeká na zprávu od jiného procesu v téže skupině
  - Může dojít ke stárnutí
    - Dva procesy si vzájemně posílají zprávy, zatímco třetí proces čeká na zprávu nekonečně dlouho
- Sdílení prostředků – problém soupeření či souběhu (*race condition*)
  - Procesy používají a modifikují sdílená data
  - Operace zápisu musí být vzájemně výlučné
  - Operace zápisu musí být vzájemně výlučné s operacemi čtení
  - Operace čtení (bez modifikace) mohou být realizovány souběžně
  - Pro zabezpečení integrity dat se používají kritické sekce

3. Producent – Konzument

Dva procesy sdílejí společnou paměť (buffer) pevné velikosti N položek

- Jeden proces je „producent“ – generuje (produkuje) nové položky a ukládá je do vyrovnávací paměti
- Paralelně běží proces „konzument“,který data vyjímá a spotřebovává



Použití:

- Obslužný program čte data ze zařízení a hlavní program je zpracovává
- Procesy mohou běžet různými rychlostmi => musí být zabezpečeno, aby nedošlo k přetečení/podtečení:

1. Implementace

Buffer omezené velikosti N je potřeba hlídat :

- Když je prázdný
  - Hlídá ho semafor e (empty); počáteční hodnota **N** (vše volno)
  - Konzument nemůže konzumovat položku z bufferu
- Když je plný
  - Hlídá jej semafor f (full); počáteční hodnota **0** (nic nezaplněno)
  - Producent nemůže ukládat položku do bufferu
- Vlastní operace s bufferem
  - Hlídá ho **binární semafor m**; počáteční hodnota **1** (volná krit. sekce)
  - Nemusí být jen pole, takže obecně vlastní vložení a výběr z bufferu ošetřit jako přístup do kritické sekce
  - Zde stačí binární semafor – kritická sekce je 0 obsazeno, 1 volno

```
Semaphore e = N; //empty
Semaphore f= 0; //full
Semaphore m= 1; //mutex
// kód producenta
While (1) {
  .. Naměřím třeba teplotu vzduchu
  p(e); // je vůbec nějaká prázdná položka, abych měl kam ukládat?
     p(m); vlozim_zaznam;v(m);
  v(f);// zvýšímpočet plných položek
}

// kód konzumenta
While (1) {
  p(f); // je tam vůbec nějaký záznam v bufferu? Snížím počet plných
    p(m);vyberu_zaznam; v(m);
  v(e);// zvýšímpočet prázdných položek
}
```

4. Čtenáři a písaři

Čtenáři a písaři

- Úloha: Několik procesů přistupuje ke společným datům
  - Některé procesy data jen čtou – čtenáři
  - Jiné procesy potřebují data zapisovat – písaři
  - Souběžné operace čtení mohou čtenou strukturu sdílet
    - Libovolný počet čtenářů může jeden a tentýž zdroj číst současně
  - Operace zápisu musí být exklusivní, vzájemně vyloučená s jakoukoli jinou operací (zápisovou i čtecí)
    - V jednom okamžiku smí daný zdroj modifikovat nejvýše jeden písař
    - Jestliže písař modifikuje zdroj, nesmí ho současně číst žádný čtenář
- Dva možné přístupy
  - Přednost čtenářů
    - Žádný čtenář nebude muset čekat, pokud sdílený zdroj nebude obsazen písařem. Jinak řečeno: Kterýkoliv čtenář čeká pouze na opuštění kritické sekce písařem.
    - Písaři mohou stárnout
  - Přednost písařů
    - Jakmile je některý písař připraven vstoupit do kritické sekce, čeká jen na její uvolnění (čtenářem nebo písařem). Jinak řečeno: Připravený písař předbíhá všechny připravené čtenáře.
    - Čtenáři mohou stárnout

5. Večeřící filozofové

- 5 filosofů sedí kolem kulatého stolu
- Každý filosof má před sebou talíř se špagetami
- Mezi každými dvěma talíři je vidlička
- Špagety jsou tak klouzavé, že filosof potřebuje 2 vidličky, aby mohl jíst
- Když filosof dostane hlad, pokusí se vzít 2 vidličky; pokud uspěje, nějakou dobu jí, pak položí vidličky a pokračuje v přemýšlení
- **Problémy:**

- **uvíznutí** = všichni filozofové zvednou levou vidličku, žádný z nich už nemůže pokračovat
- **vyhladovění** = pokud by filozofové vzali najednou levou vidličku, budou běžet cyklicky – vidí, že pravá není volná, položí…

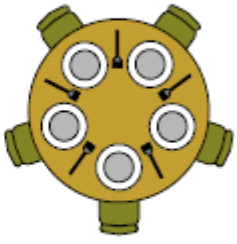
Problém večeřících filozofů

Sdílená data
- semaphore chopStick[ ] = new Semaphore[5];

Inicializace
- for(i=0; i<5; i++) chopStick[i] = 1;

Implementace *i*-tého filozofa:

```
do {
    chopStick[i].wait;
    chopStick[(i+1) % 5].wait;
        eating();          // Teď jí
    chopStick[i].signal;
    chopStick[(i+1) % 5].signal;
        thinking();        // A teď přemýšlí
} while (TRUE) ;
```



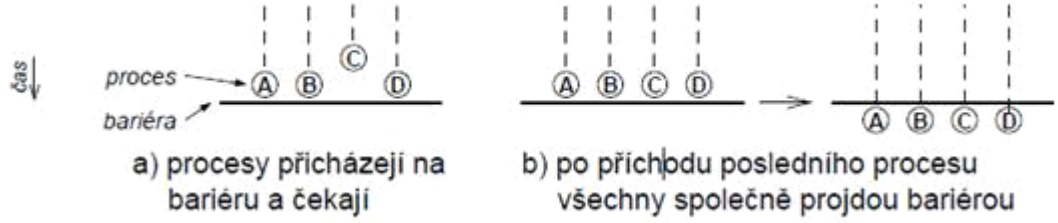
- Toto řešení nepočítá s žádnou ochranou proti uváznutí
  - Rigorózní ochrana proti uváznutí je značně komplikovaná

Filozofové, ochrana proti uváznutí

- Zrušení symetrie úlohy
  - Jeden z filozofů bude levák a ostatní praváci
    - Levák se liší pořadím zvedání vidliček
- Jídlo se n filozofům podává v jídelně s n+1 židlemi
  - Vstup do jídelny se hlídá čítajícím semaforem počátečně nastaveným na kapacitu n
  - To je ale jiná úloha
- Filozof smí uchopit vidličky pouze tehdy, jsou-li obě (tedy ta vpravo i vlevo) volné
  - Musí je uchopit uvnitř kritické sekce
  - Příklad obecnějšího řešení – tzv. skupinového zamykání prostředků

6. Bariera

- Bariéry jsou synchroniuzační mechanismus pro skupiny procesů, používají se zejména v oblasti vědecko-technických výpočtů
- Aplikace se skládá z fází, žádný proces nesmí do následující fáze, dokud všechny procesy nedokončily fázi předchozí
- Na konci každé fáze proces synchronizaci na bariéře (volá „barrier“), ta volající pozastaví, dokud všechny spolupracující procesy také nezavolají „barrier“
- Všechny procesy opouštějí bariéru současně.



Užití např. při iteračních výpočtech…

Plánování úloh a procesů v dávkových systémech

Sunday, June 2, 2013 9:19 PM

Plánovač určí, který proces (vlákno) by měl běžet nyní.
Dispatcher provede vlastní přepnutí z aktuálního běžícího procesu na nově vybraný proces.

Základní stavy procesu: (tyhle stavy jsou určené pro interaktivní systémy)

- Běžící
- Připraven (čeká na CPU)
- Blokován (čeká na zdroj nebo zprávu)
- Nový (proces byl právě vytvořen)
- Ukončený (proces byl ukončen)

Charakteristika:

- Spustit další úlohu, nechat ji běžet až do konce
- Uživatel s úlohou nekomunikuje, zadá jen program plus vstupní data
- O výsledku je uživatel informován, např. emailem

Non-preemptivní plánování:

- Každý proces dokončí svůj CPU burst
- Proces si podrží kontrolu nad CPU, dokud se jí nevzdá (I/O čekání, ukončení)
- Lze v dávkových systémech, není příliš vhodné pro time-sharingové (se sdílením času)

Plánovač (obecně):

- Tři základní charakteristiky plánovače:
  - **Rozhodovací mód** = okamžik, kdy jsou vyhodnoceny priority procesu a vybrán proces pro běh
    - Nepreemptivní:
      - Proces využívá CPU, dokud se jej sám nevzdá
      - Jednoduchá implementace
      - Vhodné pro dávkové systémy
      - Nevhodné pro interaktivní a RT systémy
  - **Prioritní funkce** pak určuje prioritu procesu v systému
    - **Dvě složky:**
      - **Statická** – přiřazená při startu procesu
      - **Dynamická** - dle chování procesu (jak dlouho čekal, aktuální zatížení systému, paměťové požadavky,..)
  - **Rozhodovací pravidla** říkají, jak rozhodnou při stejné prioritě (náhodný výběr, cyklické přidělování kvanta, chronologicky FIFO)
- Cíle plánovacích algoritmů:
  - Společné cíle
    - Spravedlivost – srovnatelné procesy srovnatelně obsloužené
    - Vynucovat stanovená pravidla
    - Efektivně využít všechny části systému
    - Nízká režie plánování
  - **Dávkové systémy**
    - dlouhý čas, omezí se přepínání úloh
    - Průchodnost (throughput)

        - Počet úloh dokončených za časovou jednotku
    - Průměrná doba obrátky (turnaround time)
        - Průměrná doba od zadání úlohy do systému až po dokončení úlohy
    - Využití CPU

    - Maximalizace průchodnosti nemusí nutně maximalizovat dobu obrátky
    - Př.
        - Dlouhé úlohy následované krátkými
        - Upřednostňování krátkých
        - Dobrá průchodnost
        - Dlouhé úlohy se nevykonají - doba obrátky bude nekonečná

1. Plánování úloh v dávkových systémech

1. FCFS (First Come First Served)

- Nepreemptivní FIFO
    - Nově příchozí na konec fronty
    - Úloha běží, dokud neskončí, poté vybrána další ve frontě
- Základní verze algoritmu předpokládá, že po dobu I/O operace je úloha zablokována a CPU se nevyužívá => řešení pokud úloha provádí I/O, zablokuje se, po dokončení I/O je zařazena na konec fronty (jako nově příchozí)
- Nevýhoda (za cenu jednoduchosti): vstupně/výstupně vázané úlohy jsou silně znevýhodněny před výpočetně vázanými

2. SJF (Shortest Job First)

- Nejkratší úloha jako první
- Předpoklad – předem známe přibližnou dobu vykonávání úloh
- **Nepreemptivní:**
    - Jedna fronta příchozích úloh - plánovač vybere vždy úlohu s nejkratší dobou běhu
- Optimalizuje dobu obrátky

3. SRT (Shortest Remaining Time)

- Úlohy mohou přicházet kdykoli
- **Preemptivní**
    - Plánovač vždy vybere úlohu, jejíž zbývající doba běhu je nejkratší
- Př.
    - Právě prováděné úloze zbývá 10 minut, do systému přijde úloha s dobou běhu 1 minutu – systém prováděnou úlohu pozastaví a nechá běžet novou úlohu
- **Problém: Možnost vyhladovění dlouhých úloh**

4. Multilevel Feedback

- N prioritních úrovní - každá úroveň má svojí frontu úloh
- Úloha vstoupí do systému s **nejvyšší prioritou**
- Na každé prioritní úrovni
    - Stanoveno maximum času CPU, který může úloha obdržet
    - Např. T na úrovni n, 2T na úrovni n-1 atd.
    - Pokud úloha překročí tento limit, její priorita se sníží
    - Na nejnižší prioritní úrovni může úloha běžet neustále nebo lze překročení určitého času považovat za chybu
    - **Proces obsahuje nejvyšší neprázdnou frontu**
    - Výhoda – rozlišuje mezi I/O vázanými a CPU-vázanými úlohami, upřednostňuje I/O vázané

algoritmus	Rozhodovací mód	Prioritní funkce	Rozhodovací pravidlo

FCFS	Nepreemptivní	P(r) = r	Náhodně
SJF	Nepreemptivní	P(t) = -t	náhodně
SRT	Preemptivní (při příchodu úlohy)	P(a, t) = a – t	FIFO nebo náhodně
MLF	Nepreemptivní	Viz popis ☺	FIFO v rámci fronty

Plánování procesů v interaktivních systémech

Sunday, June 2, 2013 9:19 PM

Plánovač určí, který proces (vlákno) by měl běžet nyní.
Dispatcher provede vlastní přepnutí z aktuálního běžícího procesu na nově vybraný proces.

Interaktivní systémy - důležitá je doba odpovědi, tj. čas od požadavku do obdržení výsledku => plánovač by měl minimalizovat dobu odpovědi pro interaktivní procesy (ale pozor přepnutí mezi procesy je drahé – brát v úvahu efektivitu)

Charakteristika:

- potřeba docílit, aby proces neběžel příliš dlouho (kvůli obsloužení dalších procesů)
- každý proces jedinečný => nepredikovatelný –neznáme čas výpočtu
- vestavěný systémový časovač v počítači – provádí pravidelně přerušení (tiky), vyvolává se obslužný podprogram v jádře, rozhodnutí o (ne)pokračování procesu

Základní stavy procesu:

- Běžící
- Připraven (čeká na CPU)
- Blokován (čeká na zdroj nebo zprávu)
- Nový (proces byl právě vytvořen)
- Ukončený (proces byl ukončen)

Správce procesů – udržuje tabulku procesů, o každém si vede záznam - z**áznam o konkrétním procesu – PCB** – Process Control Block – souhrn dat potřebných k řízení procesů (znovu rozběhnutí)

Dispatcher – předává řízení procesu vybraného short time plánovačem, provede:

- Přepnutí kontextu
- Přepnutí do user modu
- Skok na vhodnou instrukci daného programu

Více připravených procesů k běhu – plánovač vybere, který spustí jako první

Plánovač procesů (scheduler), používá plánovací algoritmus (scheduling algorithm)

Systémy se sdílením času

- Můžeme mít procesy běžící na pozadí
- Interaktivní procesy – komunikují s uživatelem

Chceme:

- Přednost interaktivních procesů

Během vykonávání procesu:

- **CPU burst** (vykonávání)
- **I/O burst** (čekání)
- Střídání těchto fází
- Končí CPU burstem
- Typicky hodně krátkých CPU burstů, málo dlouhých

Preemptivní plánování:

- **Proces lze přerušit KDYKOLIV během CPU burstu** a naplánovat jiný (-> problém kritických sekcí)
- Dražší implementace kvůli přepínání procesů (režie)
- Vyžaduje speciální hardware – timer (časovač)

Část výkonu systému spotřebuje režie nutná na přepínání procesů. K přepnutí na jiný process také může dojít v nevhodný čas (ošetření kritické sekce). Preemptivnost je ale u současných systémů důležitá, pokud potřebujeme interaktivní odezvu systému. Časovač tiká (generuje přerušení), a po určitém množství tiků se určí, zda procesu nevypršelo jeho časové kvantum.

U preemptivního pánování je nutné koordinovat přístup ke sdíleným datům.

Preempce jádra OS

- Přeplánování až ve chvíli, kdy se manipuluje s daty (I/O fronty) používanými jinými funkcemi jádra.

Plánovač

- Tři základní údaje charakterizující plánovač:
  - Rozhodovací mód = okamžik, kdy jsou vyhodnoceny priority procesu a vybrán proces pro běh
  - Prioritní funkce pak určuje prioritu procesu v systému
  - Rozhodovací pravidla říkají, jak rozhodnou při stejné prioritě
- **Rozhodovací mód**
  - **Preemptivní**
    - Kdy?
      - **Periodicky – časové kvantum** (interaktivní systémy)
    - Náklady na přepínání procesů a logiku plánovače
  - **Prioritní funkce**
    - Funkce bere v úvahu parametry procesu a systémové parametry
    - Určuje prioritu procesu v systému
    - Externí priority (třídy uživatelů, systémové procesy)
    - Priority odvozené od chování procesu (dlouho neběžel, čekal)
    - Většinou dvě složky:
      - **Statická** – přiřazena při startu procesu
      - **Dynamická** - Dle chování procesu (dlouho čekal aj.)
    - Co všechno může vzít v úvahu prioritní funkce?
      - Čas, jak dlouho využíval CPU
      - Aktuální zatížení systému
      - Paměťové požadavky procesu
      - Čas, který strávil v systému
      - Celková doba provádění úlohy (limit)
      - Urgence (Real Time systémy)
  - **Cíle plánovacích algoritmů**
    - Interaktivní systémy
      - Interakci s uživatelem, tj. I/O úlohy
      - Minimalizace doby odpovědi versus efektivita – drahé přepínání mezi procesy
  - **Plánování procesů** v interaktivních systémech
    - Potřeba docílit, **aby proces neběžel příliš dlouho**
      - Možnost obsloužit další procesy
    - Každý proces – jedinečný a **nepredikovatelný**
      - Nelze říct, jak dlouho poběží, než se zablokuje
    - **Vestavěný systémový časovač**

- Provádí pravidelně přerušení (tiky časovače, clock ticks)
- Vyvolá se obslužný podprogram v jádře
- Rozhodnutí, zda proces bude pokračovat, nebo se spustí jiný (preemptivní plánování)

1. Round Robin - Algoritmus cyklické obsluhy

- Jeden z nejstarších a nejpoužívanějších
- Každému procesu přiřazen **časový interval = časové kvantum**, po které může běžet
- Na konci kvanta je pak naplánován a spuštěn další připravený proces (preempce)
- Pokud proces skončí nebo se zablokuje před uplynutím kvanta, je naplánován a spuštěn další proces
- Jednoduchá implementace plánovače
  - Plánovač udržuje seznam připravených procesů
  - Při vypršení kvanta/zablokování se vybere další proces
  - Při **vypršení kvanta** je procesu **nedobrovolně** odebrán procesor, proces přejde do stavu **připravený**
  - **Při zablokování** - proces žádá I/O a **dobrovolně** se vzdá CPU, přejde do stavu **blokovaný**
- P**roblém**
  - V systému výpočetně vázané i I/O vázané úlohy
  - Výpočetně vázané většinou kvantum spotřebují
  - I/O vázané využijí pouze malou část kvanta a pak se zablokují
  - Výpočetně vázané úlohy tedy získají nespravedlivě velkou část času CPU
  - => modifikace VRR (Virtual Round Robin) – procesy po dokončení I/O mají přednost před ostatními

2. Prioritní plánování

- Předpoklad RR: všechny procesy jsou stejně důležité
- Přidává prioritu procesů
- Prioritu lze přiřadit staticky nebo dynamicky
  - **Staticky** – při startu procesu
  - **Dynamicky** – např. přiřadit I/O větší prioritu
- Priorita má statickou a dynamickou složku – výsledná priorita vznikne součtem obou složek
- Statická je přiřazena při startu procesu a dynamická v závislosti na chování procesu v poslední době
- Plánovač snižuje dynamickou prioritu běžícího procesu při každém tiku časovače, až priorita daného procesu klesne pod prioritu jiného procesu, dojde k přeplánování

3. Spravedlivé sdílení

- Přidělovat čas každému uživateli (či jinak definované skupině procesů) proporcionálně, bez ohledu na to, kolik má procesů
- Máme-li N uživatelů, každý dostane 1/N času

4. Plánování pomocí loterie

- Procesy obdrží tikety (losy)
- Plánovač vybere náhodně jeden tiket
- Vítězný proces obdrží cenu – 1 kvantum času CPU
- Důležitější procesy – více tiketů, aby se zvýšila šance na výhru
- Zatím spíše experimentální algoritmus

algoritmus	Rozhodovací mód	Prioritní funkce	Rozhodovací pravidlo
RR	Preemptivní vyprší kvantum	P() = 1	Cyklicky
Prioritní	Preemptivní P jiný > P	Viz text	Náhodně, cyklicky

Spravedlivé	Preemptivní P jiný > P	P(p,g) = p – g	Cyklicky
Loterie	Preemptivní vyprší kvantum	P() = 1	Dle výsledku loterie

Pozn. V prednasce ZOSzos07_2010 (cca v polovině) ještě zmiňuje plánování v RT systémech

Správa hlavní paměti, metody přidělování paměti, virtuální paměť

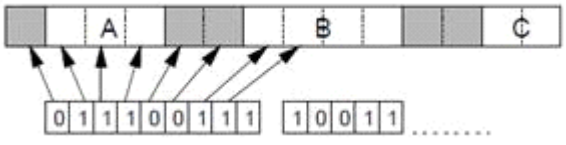
Sunday, June 2, 2013 9:20 PM

1. Část OS, která spravuje paměť, se nazývá správce paměti

- Udržuje informaci, které části paměti se používají a které jsou volné
- Alokuje paměť procesům podle potřeby
- Zařazuje paměť do volné paměti po uvolnění procesem

2. Základní mechanismy pro správu paměti

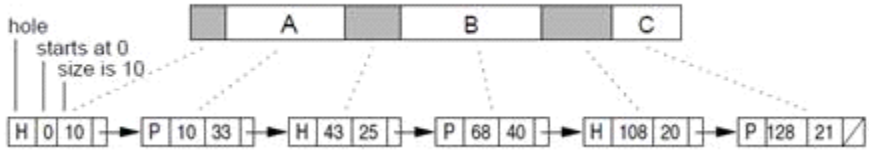
- **Jednoprogramové systémy bez odkládání a stránkování**
- Nejjednodušší – spouštíme pouze jeden program v jednom čase
- Dovoluje procesu použít veškerou paměť, kterou nepotřebuje OS
- Po skončení procesu je možné spustit další proces
- **Multiprogramování**
- **Multiprogramování s pevným přidělením paměti**
- Nejjednodušší schéma = rozdělit paměť na n oblastí (mohou být i různé velikosti)
- V historických systémech se provádělo ručně při startu zdroje
- Po načtení úlohy je obvykle část oblasti nevyužitá
- Snaha umístit úlohu do nejmenší oblasti, do které se vejde
- Několik strategií:
- Více front, každá úloha do fronty nejmenší oblasti, kam se vejde
- Jedna fronta – po uvolnění oblasti z fronty vybrat největší úlohu, která se vejde
- **Multiprogramování s proměnnou velikostí oblasti**
- Každé úloze je přidělena paměť podle požadavku
- Obsazení paměti se postupně mění, jak úlohy přicházejí a končí
- Zlepšuje využití paměti
- Postupem času může vzniknout mnoho malých volných oblastí
- OS musí vědět, která paměť je volná a která alokovaná
- Nejpoužívanější způsoby zajištění správy paměti:
    - **Správa paměti pomocí bitových map**
        - Paměť rozdělena do alokačních jednotek stejné délky
        - S každou alokační jednotkou sdružen jeden bit (0 = volno, 1 = obsazeno)
        - menší alokační jednotky = větší bitmapa
        - Větší alokační jednotky = více nevyužité paměti, protože velikost procesu nebude přesně násobek alokační jednotky
        - Výhoda – konstantní velikost bitmapy
        - Nevýhoda – pokud požadujeme úsek paměti velikosti N alokačních jednotek, musí se v bitmapě vyhledat N po sobě následujících nulových bitů (drahá operace)



    - **Správa paměti pomocí seznamů**
- Myšlenka udržovat seznam alokovaných a volných oblastí
- Každá položka seznamu obsahuje:
    - Informaci, zda se jedná o proces nebo díru
    - Počáteční adresu oblasti
    - Délku oblasti
- Práce se seznamem:

- Pokud proces skončí, nahradí se dírou
- Pokud jsou vedle sebe dvě díry, sloučí se

- Seznam je dobré mít seřazený podle počáteční adresy oblasti



  - Způsoby alokování paměti spravované pomocí seznamu:

Algoritmus first fit

- Prohledávání seznamu, dokud se nenajde první dostatečně velká díra
- Rychlý

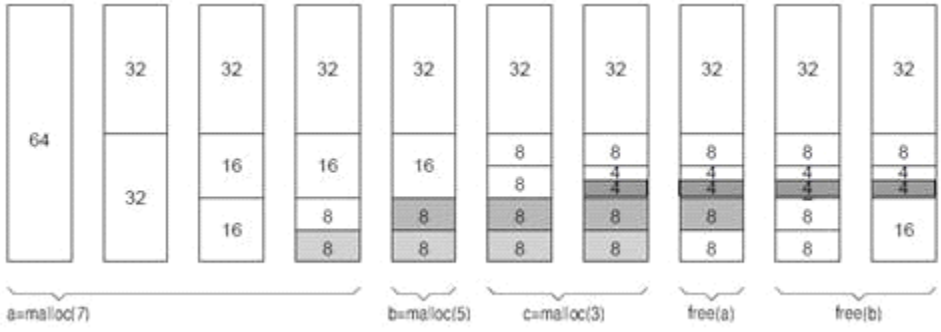
Algoritmus next fit

- Prohledávání začne tam, kde skončilo předchozí

Algoritmus best fit

- Prohledá celý seznam, vezme nejmenší díru, do které se proces vejde
- Pomalejší, protože prohledává celý seznam

  - **Mechanismus „buddy system“**
    - Mějme seznamy volných bloků velikosti 1,2,4,8,16 … alokačních jednotek až do seznamu bloků velikosti celé paměti
    - Na začátku veškerá paměť volná, všechny seznamy jsou prázdné kromě seznamu obsahující 1 položku velikosti paměti
    - Přijde-li požadavek, zaokrouhlí se nahoru na mocninu 2
    - Blok se rozdělí na 2 bloky, pokud ještě moc velké, jeden z nich se zase rozdělí na 2 bloky atd.
    - Uvolnění paměti: pokud jsou volné oba sousední bloky stejné velikosti, spojí se do jednoho
    - Neefektivní ve využití paměti, ale rychlý



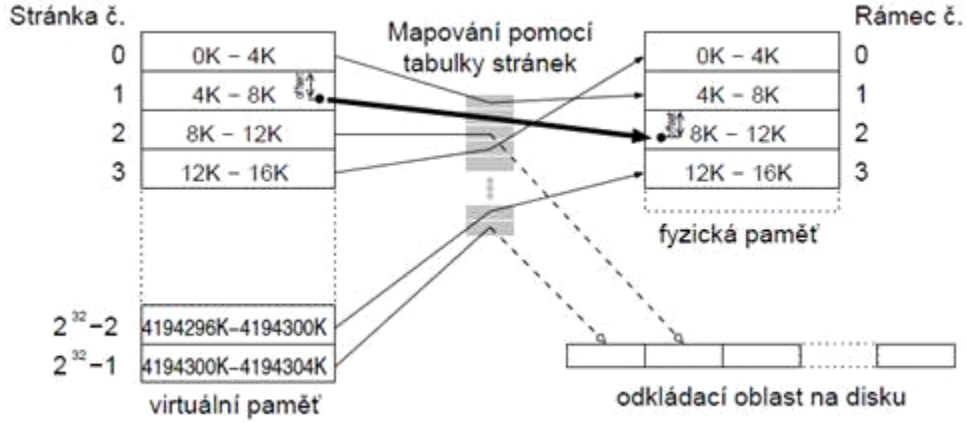
- **Relokace a ochrana paměti při multiprogramování**
  - 2 problémy:
    - Program poběží na **různých adresách** v paměti – relokace
    - Paměť programů musí být chráněna před zasahováním jiných programů – ochrana
  - Relokace při spuštění programu (zavedení programu do paměti)
    - Po kompilaci programu jsou adresy funkcí, proměnných, atd. vztaženy k počáteční adrese 0 (relativní adresy, jsou to v podstatě virtuální adresy).
    - Následně spuštěný proces programu dostane alokovaný blok paměti začínající na nějaké adrese, např. 1000. Tzn. všechny adresy funkcí, proměnných, atd. tohoto programu budou vztaženy k počáteční adrese 1000.

    ➔ Zavedení jednotky **MMU** obsahující **bázi** (poč. adresa alokovaného bloku ve fyzické paměti) a **limit** (velikost alokovaného bloku paměti). MMU zajišťuje

převod **relativní adresy v programu na adresu fyzické paměti** prostřednictvím báze. Navíc zajišťuje ochranu prostřednictvím limitu – pokud relativní adresa překročí limitní velikost alokované paměti -> vyjímka.

3. Virtuální paměť

- Problém, že programy jsou větší než dostupná fyzická paměť
- Chceme, aby ve skutečné operační paměti byla realizovaná pouze část adresového prostoru (obvykle ta část, která je momentálně potřeba), zbytek může být odložen na disku
- Procesor používá tzv. virtuální adresy
- Pokud je požadovaná část virtuálního paměťového prostoru ve fyzické paměti, MMU převede virtuální adresu na fyzickou
- Pokud není ve fyzické paměti, OS jí musí přečíst z disku
- **Většina systémů virtuální paměti používá techniku nazývanou stránkování:**
  - **Virtuální adresový prostor je rozdělen na <u>stránky</u>** pevné délky, obvykle mocnina 2 (z důvodu předejít použití dělení – místo dělení bitový posun), nejčastěji 4KB, 512B, 8KB….
  - **Fyzická paměť je rozdělena na části stejné délky – <u>rámce</u>.**
  - Rámec obsahuje právě jednu stránku.
  - Mapování stránek na rámce probíhá pomocí tabulky stránek (uložené na známém místě v paměti) prostřednictvím stránkovacích mechanismů (*čisté stránkování, stránkování na žádost)*



1. Převod virtuální adresy na fyzickou (u stránkování):

Virtuální adresa VA
Fyzická adresa FA
Číslo stránky str
Offset offset (pozice ve stránce)
Číslo rámce ramec

Předpokládejme velikost stránky 4096 bytů.

1) **Rozdělení virtuální adresy na číslo stránky a offset**

   str = VA / 4096 (celočíselné dělení)
   offset = VA mod 4096 (zbytek po dělení)

2) **Převod čísla stránky na číslo rámce prostřednictvím tabulky stránek**

   Například máme takovouto tabulku stránek (mapuje stránky na rámce):

Stránka	Rámec
0	1
1	2
2	nenamapována

Je-li VA = 5000 pak

str = 5000 / 4096 = 1 (žádné desetiny, je to celočíselné dělení)
offset = 5000 mod 4096 = 904

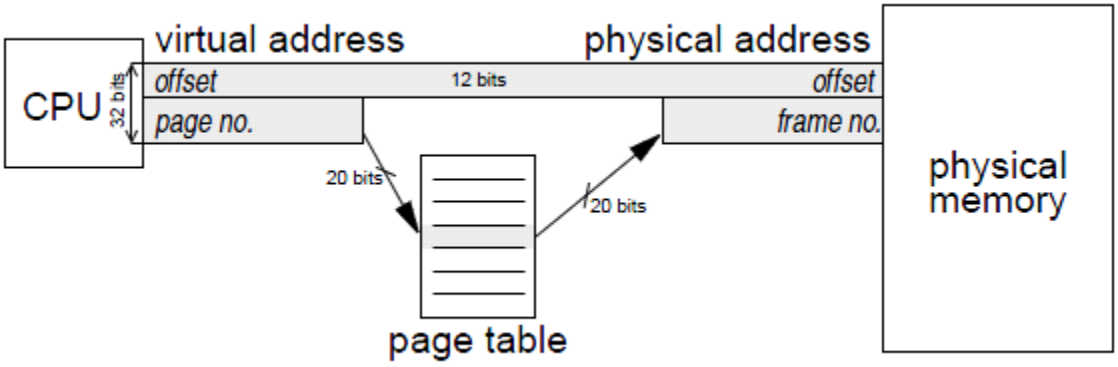
V tabulce se podíváme, který rámec odpovídá stránce 1:

rámec = 2

3) **Získání fyzické adresy na základě čísla rámce a offsetu**

FA = rámec * 4096 + offset = 2 * 4096 + 904 = 9096

Schéma realizace MMU pro stránkování paměti:



Pokud požadovaná stránka není mapována na rámec ve fyzické paměti, nastává **výpadek stránky** -> způsobení vyjímky -> zachycení vyjímky OS (nastává **přerušení**) -> OS zahájí **zavádění stránky** a přepne na jiný proces -> po zavedení stránky OS **aktualizuje mapovací tabulku stránek** -> proces může pokračovat.

MMU – Memory Management Unit

- Více procesů v paměti
  - Každý proces paměť pro sebe – např. od adresy 0
  - Ochrana – nemůže zasahovat do paměti ostatních procesů ani jádra
- Mezi CPU a pamětí je právě MMU
  - Program pracuje s **virtuálními adresami**
  - MMU je převede na fyzické

Algoritmy nahrazování stránek paměti

Sunday, June 2, 2013 9:20 PM

Jsou-li všechny rámce ve fyzické paměti **obsazené** a nastane-li **výpadek stránky** (požadované stránka není mapovaná na rámec), je nutné některou stránku z rámce fyzické paměti **vyhodit a nahradit** jí požadovanou stránkou. **Která** stránka bude z fyzické paměti **vyhozena**, rozhodují algoritmy *nahrazování stránek*.

1. Algoritmus FIFO

Stránky jsou vyhazovány z fyzické paměti v pořadí, ve kterém byly do fyzické paměti mapovány, tzn. je vyřazena ta stránka, která se ve fyzické paměti nachází nejdéle.

Problém: Může být vyhozena taková stránka, která je **stále často** používaná -> *bude se proto po vyhození brzy opět mapovat do fyzické paměti, což je neefektivní*.

1. Beladyho anomálie

Intuitivně nás napadne, že čím máme více rámců v paměti, tím méně výpadků stránek nastane.

Ovšem, pan Belady našel protipříklad právě pro algoritmus FIFO, který tento náš intuitivní předpoklad vyvracuje. *Tento protipříklad dokazuje, že nastávají případy, kde při mapování stránek na menší počet rámců nastane méně výpadků stránek, než při mapování na větší počet rámců*.

2. Algoritmus MIN – OPT (optimální nahrazování)

- Čistě **teoretický algoritmus** -> **nelze jej realizovat** a jedná se o **ideální algoritmus** pro nahrazování stránek. Má **nejmenší možný** počet výpadků stránek. Vznik byl iniciován objevením právě Beladyho anomálie.
- Algoritmus u každé namapované stránky ve fyzické paměti uchovává počet instrukcí, po které se k ní **nebude přistupovat**. V okamžiku výpadku stránky se vyhodí stránka s **nejvyšším počtem**, což znamená, že by měla být daná stránka použita v **nejvzdálenější budoucnosti**, než ostatní.
- Jelikož není možné vidět do budoucnosti a určit tak stránku, která bude použita v budoucnosti nejvzdálenější, nelze tento algoritmus realizovat. Je to optimální algoritmus, protože lepší není možné navrhnout.

3. Algoritmus Least Recently Used (LRU, LUR)

- Vyhazuje se stránka **nejdéle nepoužitá**.
- Vychází z předpokladu, že stránky, které se používaly v posledních několika instrukcích, se budou pravděpodobně používat i v následujících instrukcích. A ty stránky, které se dlouho nepoužívaly, pravděpodobně nebudou v nejbližší době zapotřebí.
- **Dobře se přibližuje k optimálnímu algoritmu.** Nemůže nastat Beladyho anomálnie.
- Je ovšem **obtížně realizovatelný** -> **softwarová realizace neefektivní**, cca 10 násobné zpomalení přístupu k paměti. Realizace má smysl jen **prostřednictvím HW**.
- Možné implementace:
  - Pomocí čítače v MMU, hodnoty čítače jsou uchovány v poli, jehož indexy odpovídají indexům rámců v paměti. Při výpadku stránky se vyhodí ten rámec, na jehož indexu v poli je nejnižší číslo.
  - Pomocí matice o velikosti „počet rámců“ X „počet rámců“ – pracuje s binárními hodnotami
- **Nevýhody:** zpomalení nutností aktualizace záznamu při každém čtení stránky z fyzické paměti
- Prakticky se nepoužívá

4. Algoritmus Not-Recently-Used (NRU, NUR)

- Vyhazuje stránky, které nejsou často používané
- Algoritmus **musí rozlišit** používané stránky od nepoužívaných -> **rozděluje je do tříd** na základě bitů **R** (Referenced) a **M** (Modified, nebo také Dirty) přiřazených každé stránce namapované ve fyzické paměti.
  - Při čtení stránky z fyzické paměti se **R nastaví na 1**.
  - Při zápisu do stránky se **R a M nastaví na 1**. (M = 1 říká, že se má modifikovaná stránka při

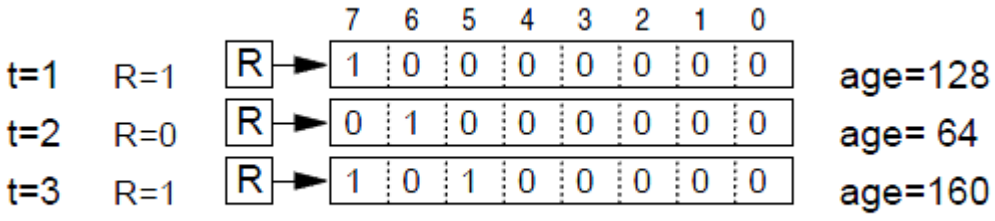
vyhození zapsat na disk)
    - Na začátku všechny stránky mají R = 0, M = 0.
    - Bit R je OS nastavován **periodicky na 0** (při přerušení časovače) -> rozlišení stránek, které byly v poslední době referencovány.
- **Třídy stránek:**
    - Třída 0: R = 0, M = 0
    - Třída 1: R = 0, M = 1
    - Třída 2: R = 1, M = 0
    - Třída 3: R = 1, M = 1
- Algoritmus NRU vyhodí stránku obsaženou **v nejnižší neprázdné třídě**. Výběr mezi stránkami spadajícími do stejné třídy proběhne náhodně.
- Často podporován přímo hardwarem
- **Nevýhody**: výkonnost (jsou i lepší algoritmy)

5. Algoritmus „Second Chance“ a „Clock“

- Vychází z algoritmu FIFO
- Provedení rozhodnutí, zda má být stránka z FIFO vyhozena, probíhá na základě bitu **R** přiděleného každé stránce ve FIFO:
    - Podívá se na **R nejstarší stránky**.
    - Pokud **R = 0**, pak je stránka **nejstarší a zároveň nepoužívaná** -> bude vyhozena
    - Pokud **R = 1**, pak stránku přesuneme **na konec FIFO** (jako by byla nově zavedena => dáme ji druhou šanci) a **R nastavíme na 0**.
- Respektive, algoritmus vyhazuje stránky, které nebyly v poslední době referencované. Pokud jdou byly v poslední době všechny stránky použité, chová se jako FIFO.
- Algoritmus CLOCK
    - Optimalizuje datovou strukturu algoritmu SecondChance.
    - Ukazuje „ručičkou“ na nejstarší stránku.
    - Při výpadku stránky:
        - Pro stránku, na kterou „ručička“ ukazuje, vyhodnotíme R.
        - Pokud R = 0, stránku vyhodíme a „ručičku“ posuneme o 1 pozici.
        - Pokud R = 1, nastavíme R na 0, ručičku posuneme o 1 pozici.

6. Algoritmus Aging

- Aproximuje algoritmus LRU
- Každé stránce je přiřazen **bit R a pole AGE** o velikosti N bitů (např. 8)
- Při přerušení časovače jsou hodnoty v poli **AGE posunuty o 1 pozici vpravo**, **zleva se přidá hodnota bitu R** a **R se nastaví na 0**.



- Při výpadku stránky se vyhodí stránka, jejíž pole **age** má **nejnižší hodnotu** (hodnota je tvořena bity obsaženými v poli).

7. Shrnutí

- **MIN – OPT optimální algorimus**
    - Nelze implementovat, pouze pro srovnávání.
- **FIFO**
    - Vyhazuje nejstarší stránku.
    - Je jednoduchý, ale trpí Beladyho anomálií (může vyhodit důležité stránky).
- **LRU**
    - Výborný.

- Obtížná implementace, vyžaduje speciální HW.
- Prakticky používán jen zřídka.

- **NRU**
  - Rozděluje stránky do 4 kategorií podle bitů R a M.
  - Není příliš efektivní, avšak občas používán.
- **„Second chance“ a „Clock“**
  - Vycházejí z FIFO
  - Nejstarším používaným stránkám dávají „druhou šanci“, berou je za nově zavedené.
  - Používané např. v některých variantách UNIXu.
- **Aging**
  - Dobrá aproximace LRU => je oproti LRU efektivní.
  - Často prakticky používaný.